CANADIAN THESES ON MICROFICHE

THÈSES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

Canada

NL 339 (r. 86/09)

The University of Alberta

INFORMATION-PRESERVING TRANSFORMATIONS OF DATABASE SCHEMAS

by

© J. Brett Hammerlindl

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Fall, 1986

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-32554-2

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: J. Brett Hammerlindl

TITLE OF THESIS: Information-preserving Transformations of Database Schemas

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1986

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

J. Brett Hammerlindl

57 Lafonde Crescent,
St. Albert, Alberta,
Canada, T8N 2N7

Date: October 6th, 1986

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Information-preserving Transformations of Database Schemas** submitted by **J. Brett Hammerlindl** in partial fulfillment of the requirements for the degree of **Master of Science.**

W.W. Armstrong

Supervisor

Gerald Cliff

Edward Ly

Date October 7, 1986

To Donna

ABSTRACT

This thesis studies database schema transformations that are information preserving. In the semilattice data model, recursive type definition and four type constructors produce schemas useful in describing complex data. Information-preserving transformations replace a schema with an informationally equivalent schema. The transformations are essential if the semilattice data model is to have one of its special features. This feature is the ability to allow each application to have its own complex view of the data.

Almost twenty basic transformation are described in this thesis. These transformations are used to build a finite Church-Rosser replacement system. This system simplifies any schema in many small steps. With this system, other results are obtained. The expressive power of the union type constructor is explored. Also some problems are shown to be NP-hard. These problems concern the equivalence of schemas with respect to the transformations given in the thesis.

Although the thesis uses notation and concepts of the semilattice data model, most of the results have analogies in other data models and many programming languages.

Acknowledgements

I thank my wife for the tremendous amount she has done for me during my graduate studies. Without Donna's encouragement, I may not have started graduate studies. Without her support and love, I would never have finished. Without her proofreading, I would still be correcting the grammar.

When I had doubts, my parents encouraged me to start and to continue my graduate studies. For this and so much more, I shall continue to love and respect them.

For his supervision of my research, I thank Professor William W. Armstrong. I appreciate the time that he spent with me, his concern for the quality of my work and his financial support of my research.

Professors Cabay, Chan, and Cliff served on my committee. I appreciate that they accommodated my schedule and that they did their duties in a thorough and considerate manner. I also thank Professor Davis and Professor Green for the pep talks they gave me.

When I needed support, assistance, or advice, I received it from many people. I thank them all and will remember that Chung Hee Hwang, Gordon Atwood, Darrell Makarenko, and Ajit Singh were especially helpful.

The Natural Science and Engineering Research Council has my gratitude for their financial support. I also thank the University of Alberta and its Department of Computing Science for supporting my research.

Table of Contents

Chapter	Page
Chapter 1: Introduction	1
1.1. Informal Definitions	2
1.2. The Research Area	3
1.3. An Outline of the Thesis	7
Chapter 2: Previous Related Work	9
Chapter 3: The Semilattice Data Model	17
3.1. A Description of the Semilattice Data Model	17
3.1.1. The Set Type Constructor	22
3.1.2. The Sequence Type Constructor	24
3.1.3. The Tuple Type Constructor	26
3.1.4. The Union Type Constructor	27
3.1.5. Definitions	28
3.2. A Comparison of the Semilattice and Relational Data Model	30
3.3. Semilattice Schema Domains	33
Chapter 4: Information-preserving Transformations	37
4.1. Definitions and Discussion	37
4.2. Non-structural Information-preserving Transformations	40
4.2.1. Substitute a Type's Definition for Its Name	41
4.2.2. Replace Two Identical Types by One Type	41
4.2.3. Change a Type's Name	41
4.2.4. Change a Tag or Attribute Name	41
4.3. Structure-changing Information-preserving Transformations	42
4.3.1. Include a Subordinate Tuple's Attributes	43
4.3.2. Include a Subordinate Union's Domains	44
4.3.3. Change a Tuple with a Union to a Union of Tuples	46
4.3.4. Eliminate a Union that is inside a Set	47
4.3.5. Simplify a Single-attribute Tuple	48
4.3.6. Remove Simple Recursion from a Type	48
4.3.7. Combine Identical Domains of a Union	50
4.3.8. Combine Similar Domains of a Union	51
4.3.9. Split an Unreferenced Union Type	52
4.3.10. Combine Two Sequences in a Tuple	53
4.4. Corrective Information-preserving Transformations	55
4.4.1. Eliminate an Unreferenced Valueless Type	56
4.4.2. Simplify a Tuple with the Empty Domain	56
4.4.3. Simplify a Set on a Valueless Type	56
4.4.4. Simplify a Sequence on a Valueless Type	57

4.4.5. Drop an Domain on a Valueless Type from a Union 57
4.4.6. Remove a Singleton Attribute from a Tuple 58
4.4.7. Change a Set on the Singleton Domain to a Boolean 58
4.5. Final Remarks 59
Chapter 5: Normalizing Semilattice Schemas 61
5.1. Replacement Systems 62
5.2. The Desired Replacement System 63
5.2.1. How the Desired Replacement System Fails to be Finite 64
5.2.2. How the Desired Replacement System Fails to be Church-Rosser 65
5.2.2.1. Substitute a Type's Definition for Its Name 66
5.2.2.2. Combine Similar Domains in a Union 66
5.2.2.3. Combine Two Sequences in a Tuple 70
5.2.3. Making a Church-Rosser Replacement System 72
5.3. A Finite Church-Rosser Replacement System 75
5.3.1. The Replacement Relation 75
5.3.2. A New Method to Prove Finiteness of a Replacement System 80
5.3.3. Proof of Finiteness 82
5.3.4. Proof of the Finite Church-Rosser Property 86
5.3.5. Improving the Replacement System 95
5.4. Interesting Results 96
5.4.1. Properties of Normal Form Schemas 96
5.4.2. The Expressive Power of the Union Constructor 97
5.4.3. The Equivalence Problems 99
Chapter 6: Conclusions 102
6.1. Review of Results 102
6.2. Proposals for Further Work 104
References 106

List of Tables

Table	Page
3.1 The Type Definitions of a Semilattice Database Schema	21
5.1 The Replacement Relation	76
5.2 The effects of the reductions on complexity	84
5.3 Cases in the proof of Sethi's P3 property	87

Chapter 1

Introduction

It is intended that the semilattice data model deal with complex data in an effective manner [Arm84]. To do this, the model requires the ability to isolate applications from the conceptual organization of the data. This has two major benefits. Application development is simplified when the designer may assume an ideal conceptual schema for the data. Secondly, application maintenance is reduced when the applications are independent of the schema since changes to the conceptual schema do not require changes to the applications.

This thesis contains work critical to allowing the semilattice data model to isolate applications from the conceptual schema. In particular, this thesis describes almost twenty basic information-preserving transformations. Some of the transformations are based on the work of others, while other transformations are original work. Typically, these transformation produces a new schema by replacing an expression in a schema definition by an equivalent expression. Each transformation includes a method of expressing the values of one schema as values of the other schema. To show the feasibility of one class of transformations, the relationship of semilattice database schemas to context-free grammars is developed.

This thesis also contains some theoretical results. It describes a method of simplifying any semilattice database schema. The method, a replacement system, simplifies a schema in many small steps. These steps are called reductions and each is based on some basic information-preserving transformations. The replacement system is shown to have the finite Church-Rosser property. Put in imprecise terms, this means that for any schema, the system always terminates and produces the same result. Since the system has this property, several useful results can be shown and an interesting conjecture made.

Although the notation and definitions used in this thesis are those of the semilattice data model, the results are of interest to those working with recursively defined data types. Data types similar to semilattice database schemas are supported by other data models and some object-oriented and strongly-typed programming languages. The reader will gain insight into the information capacity of data types that are recursively defined with the tuple, set, sequence and union type constructors. Also, the reader will be introduced to a new technique to show the termination of a replacement system and given an appreciation of the difficulty of showing that a replacement system has the Church-Rosser property.

The remainder of this chapter contains some informal definitions, a discussion of the research area, and an outline of the remainder of the thesis. The informal definitions are those of terms used in the explanation of the research area. Formal definitions of terms are in later parts of the thesis. The section on the research area describes the particular problems that were studied, the motivations for the research, and the approach that was taken.

1.1. Informal Definitions

A semilattice database schema is a finite set of data type definitions. A data type may be basic: integer, string, et cetera. Otherwise, a data type is constructed from other types with the tuple, set, sequence and union constructors. Recursive definition of types is permitted. Two schemas are informationally equivalent if there is a one-to-one correspondence between their domains. A schema transformation is any binary relation of database schemas. A transformation is information preserving if a schema is always transformed to an informationally equivalent one.

In this thesis, a database schema is in normal form if it is an irreducible element of a finite Church-Rosser replacement system. A replacement system consists of a set and a binary relation, the reduction relation, on that set. An element of the set is

reduced by replacing it by any element that the reduction relation relates to it. An element is irreducible if it can not be reduced. If a replacement system does not have an infinite series of reductions, it is finite. A replacement system is finite Church-Rosser if it is finite and all series of reductions that start with a particular element, end with the same irreducible element.

1.2. The Research Area

This thesis is modeled on Richard Hull and Chee K. Yap's study of data representation [HuY82, HuY84]. Their results included six reductions that change a schema to a simplier one with equivalent information capacity. Hull and Yap defined a replacement system based on these reductions. They proved their replacement system is a finite Church-Rosser one. Therefore, their replacement system gives a method of testing informational equivalence of schemas since two informationally equivalent schemas must have the same normal form. From their work, it is straightforward to generate a procedure to express data in any schema whenever the data is stored in any equivalent schema.

This thesis describes the results obtained by applying Hull and Yap's methods to the semilattice data model. Since their definition of equivalence is inadequate for schemas with data types having recursive definitions, the definition of equivalence is based on the work of others. The problem and the results are significantly different from Hull and Yap's because the semilattice data model is able to construct a much broader class of schemas.

This thesis describes some basic information-preserving transformations of semilattice database schemas. These transformations are the initial step of an important process. When complete, the process shall give the semilattice data model a special ability. This is the ability to hide the conceptual organization of the data from the applications that use that data. When the applications are independent of the

conceptual schema, advantages can be realized that are similar to the advantages found in hiding the physical organization of the data from the applications. One advantage is that the conceptual schema may be transformed to a compatible schema without requiring changes to the applications. A new conceptual schema might be desired because it is more efficient, versatile, or reliable. Another advantage is that application design is simplified since a designer can assume any reasonable conceptual schema.

To a degree, the universal relation data model isolates applications from the conceptual schema of the data [Ull82]. It does this by giving all applications the same structureless representation of the data. Although a structureless representation is easy to comprehend, the designers of some applications require or prefer structured views of the data. It is intended that the semilattice data model will allow each application to have its own complex view of the data.

The semilattice data model uses the basic information-preserving transformations to give each application its own complex view of the data. The transformations provide our method of showing that some schemas are informationally equivalent. This is important since an application's view of the data is compatible with the conceptual schema only if certain conditions exists. One of the necessary conditions is that the application's view is produced from a schema informationally equivalent to the conceptual one. Another critical function of the transformations is their role in producing the data for an application from the actual data. Each transformation provides a method of producing data organized according to one schema from data organized according to the other schema. The transformations' methods of transforming data can be composed into procedures. These procedures can be used to transform the actual data values of the conceptual schema into values of an application's view.

This thesis describes a finite Church-Rosser replacement system that normalizes

semilattice database schemas. The reduction relation of the replacement system is generated from a group of information-preserving transformations. A semilattice database schema is normalized by reducing it to an irreducible element of the replacement system. If possible, a schema is normalized to a collection of self-defining, non-recursive, simple data types. In other situations, the normalization process reduces recursion, references and type complexity as much as possible.

What are the motivations for normalization of semilattice database schemas? One benefit is that normalization induces a second valuable definition of schema equivalence; schemas are *normal-form* equivalent if they have the same normal form. Another closely related equivalence relation can be generated from the union of the transformations but for that relation, determining equivalence would certainly be far more computationally intensive and possibly undecidable. Given either definition of equivalence, the properties of an equivalence class are of interest.

Another advantage of normalization is that both the process of normalization and the characteristics of the normal form yield considerable insight into the expressive power of the four type constructors. For instance, it is shown in Chapter 5 that any schema is informationally equivalent to one where every data type definition is either the union of expressions with no union constructs or an expression with no union constructs. It is also shown that for any schema, it is possible to detect the types with the empty domain, remove them and preserve informational equivalence. These properties of database schemas are significant in their own right and hopefully, the tasks of forming and proving some theorems will be simplified because schemas with union subtypes and domainless types can be ignored. Other results found in studying the characteristics of the normal form could have similar benefits.

There are several advantages to normalizing with a finite Church-Rosser replacement system. Hopefully, each reduction is small, local and intuitive and therefore the

credibility of the normalization is increased. Any heuristic technique can be used to pick the next reduction to apply since there is never any risk of an infinite sequence of reductions or any significant effect on the final outcome. Also when it is known that a group of reductions has no two with overlapping effects, all may be done in parallel. With a finite Church-Rosser replacement system, normal-form equivalence is a decidable for two schemas. This may provide a method of testing informational equivalence of schemas as it does in Hull and Yap's replacement system [HuY82, HuY84].

It was decided not to include constraints in the schema other than those that follow from the structure. The main reason for doing so is that schemas are not constrained in the semilattice data model. Another reason for omitting constraints was to focus the work on structure. A goal of this thesis is to understand the significance of the conceptual structure of a database schema since relatively little work has been done on this topic. Even without constraints, the problem was challenging and the results enlightening.

The original plan was to find as many basic information-preserving transformations of the semilattice database schemas as possible and then to build a finite Church-Rosser replacement system that incorporates all those found. This system would reduce all informationally equivalent schemas to a common normal form. The replacement system would show that informational equivalence is a decidable property and would provide the tools to navigate from any given schema to any equivalent one.

In time, it became evident that this ultimate replacement system is probably impossible to build because of the rich diversity of semilattice database schemas. Nevertheless, it is possible to build a powerful finite Church-Rosser replacement system that incorporated most of the information-preserving transformations that are documented in this thesis. This was done and some useful results were obtained. These results are reported in this thesis.

1.3. An Outline of the Thesis

In Chapter 2, previous work related to this research is reviewed. Mainly, this describes the work of others on schema equivalence and transformations of database schemas.

Chapter 3 deals with the semilattice data model. The first section is a description of the semilattice data model. Next is a section that compares the relational and the semilattice data models. The concluding section of the chapter relates semilattice database schemas to context-free grammars. This relationship provides a technique to detect a type definition that has no domain.

Basic information-preserving transformations of the semilattice data model are described in Chapter 4. The first section precisely defines informational equivalence and information-preserving transformation. The body of the chapter describes three categories of information-preserving transformations. The *non-structural* ones are those that change some aspect of the schema that is not related to the structure of the data. The *structure-changing* information-preserving transformations apply to a reasonably designed database. The *corrective* transformations apply to syntactically correct schemas that have constructs with little or no information content. In the final remarks, the reasons for not accepting some other transformations as information preserving are given.

Chapter 5 describes the two replacement systems that were developed. First, precise definitions of the terms used to describe replacement systems are given. The chapter then defines a replacement system based on all of the transformations of Chapter 4. Then it is shown that the system is neither finite nor Church-Rosser. This is done by giving examples of schemas that the system normalizes inappropriately. These examples are followed by a description of the attempts to correct the problems. Next, a second replacement system is defined and its possession of the finite Church-

Rosser property is proven. The proof of finiteness is based on an interesting new technique developed for this thesis. The chapter concludes with some observations and results.

A summary of the major results of the thesis and some proposals for further work are in the final chapter.

Chapter 2

Previous Related Work

This chapter reviews some publications concerned with issues that are relevant to this thesis. Other than Hull and Yap's work, little could be found that was directly concerned with the topic of this thesis. Some articles on database schema equivalence and general schema transformations were located. Since these articles included some relevant points, they are reported below.

E. F. Codd [Cod70] proposed the relational model and described the best known database schema transformation, normalization. This transforms any schema to one in *first normal form*. Essentially, normalization transforms a schema to a related one that has no composite attributes and no physical navigation. The intention is to preserve the information content while putting the data in a useful form that is machine independent and mathematically eloquent. Codd granted that normalization is applicable only to applications that have certain properties but he could not conceive of an application where the necessary conditions did not exist. The conditions are that the primary keys are an aggregation of simple domains and the interrelationships between non-simple domains are hierarchical.

Codd reported further schema transformations in [Cod72]. These transformations use functional dependencies to determine a set of relation schemas from a single relation schema. The set of schemas implicitly enforce the same constraints that the original schema had to enforce explicitly. A set of projections is the corresponding data transformation. The relations produced by the projections are less redundant than the original relation but they have the same information.

Codd's work inspired tremendous efforts that found better algorithms to transform a relation in first normal form to a less redundant schema that enforces the same constraints. Maier's text [Mai83] contains a thorough coverage of this area.

Most of the work in this area is based on the universal relation concept. Jeffrey Ullman's paper [Ull82] presented an excellent explanation and defense of the universal relation concept. Ullman describes a half dozen different universal relation assumptions that have been published. The universal relation schema assumption is the simpliest and is part of each of the other assumptions. This assumes that if "we have done sufficient renaming of the attributes that a unique relationship exists among any set of attributes." Thus it is meaningful to talk about the universal relation schema. This hypothetical relational schema has all the attributes of all the schema in the database.

Supporters of the universal instances assumption made conjectures that it is possible to design a database schema for any application that supports the assumption. This assumption holds that at any time an application's database is the set of the appropriate projections of a relation of the appropriate universal relation schema. A definition of database schema equivalence, based on this assumption, was presented in [BMS81]. Two schemas are equivalent if their universal relations have the same schema and they *faithfully represent* the same relations of that schema. To faithfully represent a relation, a join of the projections of the relation onto the schema must produce the original relation. Naively, a database in one schema is transformed to one in the other schema by forming the universal relation by joining all relations of the database and then doing the appropriate projections. Realistically, algebraic techniques should be used to minimize the amount of processing required. For this definition of equivalence, testing schema without constraints for equivalence is a trivial matter. Two schemas are equivalent if both schema have the property that the attributes of each component are a subset of the attributes of some component of the other schema. A refined definition of equivalence allows explicit constraints. If these are functional and join dependencies, testing equivalence requires the exponential *chase* computation but is decidable.

In contrast, Tim Connors [Con85] presented a definition of equivalence of different views of the same relational schema. The query capacity of a view is the set of all queries of the original database schema that some query of the view will answer correctly for all database states. Two views are equivalent if their query capacity is equal. He showed that this is a decidable property.

Sheldon Borkin [Bor80] studied equivalence of data models, their application models, and database states. He required one-to-one correspondence between various sets in his definition of equivalence for database states. He insisted that for two application models to be equivalent the operations of each must be expressible by the other. His treatment included schemas with explicit constraints and null values. To do so required a partial ordering of tuples by an information measure. In conclusion, a "dual" semantic data model was suggested. The semantic relational model would be used for queries and the semantic graph (network) model for design of the schema and updates of the data.

Y. Edmond Lien [Lie82] showed the equivalence of parts of the relational and network data models. This was done by giving two functions. These mapped a constrained schema in one model to the appropriate one in the other model. Restrictions on the types of constraints apply to both models. A network schema must have a loop free Bachman diagram. This means that no existence requirement can run from an entity to itself. Relational schemas are limited to those with contention-free and conflict-free multivalued dependencies.

Atzeni, Ausiello, and Batini [AAB82] defined *weak* and *strong inclusion* between schemas of any data model with respect to a query language of the model. A schema is weakly included in another if there is a query that takes any value of the schema to a value of the other. A schema is strongly included in another if there are two queries such that the first takes any value of the schema to a value of the other and the com-

position of the queries is the identity mapping on the first schema. Two schemas are *equivalent* if they are each strongly included in the other. Thus schemas are equivalent if and only if there is one-to-one function from the set of database states of one schema onto the set of database states of the other schema.

Most research in the database area has been based on the relational data model and universal relation assumptions. However, the assumptions of this research have been criticized [AtP82]. First normal form is not desirable for a temporal database [GaV85] and in other situations [SAH84]. Articles have discussed the difficulty of using the relational model for computer aided design applications [Lor82, ScP82] and for statistical applications [ShW85]. Many new data models have been proposed [ACO85, Arm84, FoV82, HuY84, KuV85, ZaH85]. Authors have also proposed various ways of extending the relational data model [AbB84, AMM83, DGK82, Har84, HNC84, JaS82, KTT83, KiK82, MUS82, OzY85, Zan83]. A review of the portions of these works that deal with schema transformations and equivalence follows.

Jaeschke and Schek [JaS82] introduced the nesting and unnesting operations for relations. These have related schema transformations. If a relation is nested on an attribute, the result is a relation on a schema where all tuples have a set value on the nesting attribute. All tuples of the original relation that agree on the values of all attributes but the one being nested on are replaced by a single tuple having all the information. If the definition of schema equivalence from [AAB82] is used, the nesting transformation gives an equivalent schema since every nested relation has two implicit constraints. A nested relation does not allow the empty set as value of the nesting attribute and has an implicit functional dependency that states the nested attribute is determined by the others. Unnesting, the second operation, preforms the reverse of the nesting operation on a relation. Some algebraic properties of these operations, by themselves and in conjunction with the standard ones like projection, were given. The most interesting result is that nesting a relation with a multivalued dependency on the

attribute that is the right hand side of the dependency gives a relation with a functional dependency with the same right and left hand side as the dependency of the unnested relation. Other research [OzY85, FST85, GuF86] has been based on the nesting transformation but it is not relevant to this thesis.

The *Format* model, proposed by Hull and Yap [HuY82, HuY84], gives a different notion of equivalence for a restricted class of database schemas. Schemas are called formats and each basic domain is a format. A new format (database schema) is constructed by applying one of three domain constructors to existing formats. The constructors are collection, composition and classification. Collection creates a domain whose values are any finite set of elements from the underlying domain. Composition creates a domain whose values are tuples. Each tuple has the same attributes with each attribute having a value from an associated domain. Classification creates a domain that corresponds to the union of the cartesian product of an identifier with each of the underlying domains.

There is a format for each unconstrained first normal form relation. Since a relation is a set of homogeneous tuples with atomic attribute domains, the corresponding format is a collection of the composition of some basic domains. A relational database, a group of relations, can be described by a format that is constructed by applying classification to the formats that describe the relations.

Some serious limitations are put on the formats: only hierarchical and relational schemas have related formats; explicit constraints are not permitted; and domain constructors like recursion, sequence and multi-set are not used. Later in [AbH84], Hull said that the Format model: "is too limited to serve as a "real" semantic data model." Nonetheless, it was powerful enough to obtain some significant results. A preliminary analysis indicates that with a concerted effort similar results could be obtained for a model that incorporated functional dependencies and some other constraints.

The Format model associates a function with every format and considers two formats equivalent when their associated functions are equivalent. The function associated with a format determines the number of possible data values for a format given a particular cardinality of each of the basic domains. The function is defined for all combinations of all possible cardinalities, $\{0,1,2,\cdots,\infty\}$, for each of the basic domains. This gives some implementations independence since it does not treat the formats *integer* and *real* as equivalent, even if the cardinalities of the integer and real domains are the same for a particular implementation.

Hull and Yap gave six reductions of formats and showed that these are information preserving since applying any of the reductions to a format yields an equivalent format. Each reduction induces a natural linear one-to-one function from the values of the one format onto the values of the other. They said a format is in *normal form* if no further reductions can be applied. Further, it was shown that these reductions form a finite Church-Rosser replacement system. This is significant since a normal form is always reached in a finite number of reductions and the normal form produced after applying any one of a group of possible reductions is equivalent to the normal form obtained by applying any of the other possible reductions. Therefore, a normalization algorithm may select the reduction to apply with no concerns that the choice will lead to an infinite series of reductions or affect the final outcome. Simple and unique normalization is essential to a good procedure for testing if two arbitrary formats are equivalent. Each format is taken to its normal form and then these are compared to find an isomorphism. This comparison has $O(n)$ complexity where n is the length of the expression describing the format. This follows because the normal forms may be represented as trees and the tree isomorphism problems has $O(n)$ complexity [RND77].

Hull did further work on information capacity. Four definitions of schema equivalence were given in [Hul84]. He showed the least restrictive is the one used in

the format model and the most restrictive is *calculus* equivalence. This is "a variant of" the [AAB82] definition given above. One significant result, Theorem 6.1, was that two relation schemas without dependencies are equivalent if and only if they are identical.

Gabriel Kuper and Moshe Vardi proposed the *logical* data model in [KuV84]. This has the same constructors as the Format model but uses them in a different manner. The result is that schemas of the network model as well as those of the relational and hierarchical models can be described. A database schema is an arbitrary directed graph with the appropriate labels on its nodes. When the graph is a tree, the resulting schema is the same as the corresponding schema of the format model. The logical data model allows its user to examine both the value and the location of data. Location is in a symbolic sense rather than a physical sense. Since the model permits values to refer to themselves, the user needs the locations in order to examine recursive values in a reasonable manner. The logical data model has logic-based query and constraint languages. Although these languages are powerful, an effective bottom-up procedure for the evaluation of queries and the enforcement of constraints is obtained when recursive schema definition is restricted. Thus, the expressive power of recursive schemas was of great interest to Kuper and Vardi. This was the subject of their work reported in [KuV85]. They used a definition of schema equivalence based on Hull's query equivalence that is mentioned above. They gave an algorithm for transforming most cyclic schemas to equivalent acyclic ones. As Codd did with normalization, Kuper and Vardi assumed that certain structures are unnatural and certain dependencies are always present. Although the paper has some insights into the nature of recursive schemas, it is not definitive. The authors conclude with the statement

"We believe that the issue of cycles deserve further study."

Other schema transformations of non-normalized relational models have been reported. Frequently, the authors treated structure superficially. Furthermore, using

any of the above definition of equivalence, a schema is not transformed to an equivalent one. Typically, a schema is strongly included, as defined by [AAB82], in the schema to which it is transformed.

Dayal, Goodman and Katz [DGK82] described a transformation of a schema with multi-sets to one with sets only. Ariswa, Moriaya and Miura [AMM83] described a space compression technique that transforms first normal form schemas to one with sets. Shoshani and Wong [ShW85] described transposition. This transformation replaces a set of tuples by a tuple of sequences and has large performance benefits in statistical applications.

In conclusion, the literature has limited value for the purposes of this thesis. It guides the development of a good definition of informational equivalence. It describes many schema transformations but few of these transform a schema to an informationally equivalent one. The work on the Format model offers a good approach to the problem. With respect to recursively defined data structures, nothing was found in a detailed search of the database literature. A quick review of the literature on programming languages also failed to find articles. Apparently, no research of this topic has been reported.

Chapter 3

The Semilattice Data Model

W. W. Armstrong developed the semilattice data model with the intention that it would effectively store, retrieve and manipulate large, complex objects [Arm84]. It was felt that certain deficiencies of the relational model of E. F. Codd [Cod70] could be corrected without sacrificing the relational model's ability to handle data in an abstract and eloquent manner. The first section of this chapter describes the semilattice data model. A comparison between the semilattice and the relational data models is made in the second section. The final section relates all semilattice database schemas to a subset of the context-free languages. This is significant since it proves that there are efficient techniques to detect the types of a schema with the empty domain.

More information about the semilattice data model can be found in other documents [Arm84, Arm86, Sin86, Bob85].

3.1. A Description of the Semilattice Data Model

The first part of this section is an introduction to the semilattice data model. The introduction has a semi-formal description of the data model and an illustrative example of a semilattice database schema. The introduction is intended to prepare the reader for the more formal description that is contained in the rest of the section. After the introduction, there is a sub-section on each of the four type constructors. A constructor's section describes the notation used to define types and denotes values of the type. This notation is used extensively in the remainder of the thesis and therefore it is advised that particular attention be given to it. A constructor's section also describes the operations of the data types that the constructor forms. For each operation, the section gives its *signature* and a brief description of the value it returns. This draws heavily from the work of Guttag, Horning and Wing in [GHW85]. That manual describes Larch, a state-of-the-art specification language. The description of

the semilattice data model concludes with a sub-section containing the proper definitions of some terms used throughout this thesis.

It is generally held that a data model must have certain properties. One text writes that:

> "A data model defines general rules for the specification of the structures of the data and the operations allowed on the data." (page 10 of [TsL82])

The specification of a data structure is usually accomplished by giving a database schema and a set of constraints. The schema defines the set of values that a database with that schema may have. The constraints restrict the possible values of a database to a subset of the possible values for the schema. In the semilattice data model, constraints are not associated with database schemas. Therefore, they are outside the scope of this thesis.

A semilattice database schema is a set of type schemas. Each type schema define a data type. A data type is characterized by its domain and operations. The *domain* of a data type is the set of all data values of that type. The *operations* of a data type have a variety of purposes. Some subset of them must generate all values in the domain. One operation determines if two values of the domain are equal. Other operations are defined so that any value given to a generating operation can be retrieved by some operation. Other useful operations are also included for some data types. The operations of the semilattice data model are the operations of the data types of its schemas.

A data type of a semilattice database schema may be a basic type of the model. The basic types of the model are string, integer, and so on. The domains and operations of the basic types are the traditional ones offered in a good programming language. In this thesis, the semilattice model is extended with an additional basic data type. The type *singleton* has only one value and two simple operation. The single value is denoted by the symbol γ. One operation generates this single value. To

test the equality of two tuples, the equality the value of each attribute is tested. Therefore, a trivial equality operation is defined for the *singleton* data type. This operation always return *true*. The data type *singleton* was added because it is required by some useful schema transformations.

If a data type of a semilattice database schema is not a basic type of the model, it must be a constructed type. The constructed types are formed by the set, sequence, tuple, and union type constructors from basic or constructed types. For every constructed type, both its domain and operations are determined by its type constructor and by the types used to construct it. The generating operations of a data type only use values of the constructing data types and other values of the constructed data type. Note that a restriction is placed on all operations that generate values of a type. Even when a type is defined recursively, it is required that every value in its domain be constructed in a finite number of operations. This means that it is not permitted that any value of any type properly contain itself. An analogy can be made. Consider a definition of arithmetic expressions that includes a statement that an expression can be formed from two expressions and an operation. Although this statement makes the definition recursive, it does not allow infinite expressions. In any expression, the proper sub-expressions are required to be *other* expressions and not the expression itself.

The set type constructor forms a type with a domain that is the set of all finite sets of elements of some type. The operations of a set data type are those that would be expected. These include the traditional mathematical set operations of union, intersection and set difference. There is no operation for set complement because the operation is infeasible for most data types. When a set data type has an infinite domain, the complement of every element is infinite and therefore not in the domain of the set data type. Even when the data type has a finite domain, the complement of an element is usually too large to manage reasonably. Section 3.1.1 has further discussion

of set data types.

The sequence type constructor forms a type that is the set of all **finite** sequences of elements of some type. A variety of operations are available to build and manipulate sequences. A description of the sequence data type is in Section 3.1.2.

The tuple type constructor forms a type with a domain that is the set of all tuples formed according to a finite mapping of attribute names to types. The attribute names are strings used to distinguish the components of the tuple. Two tuples are equal if the values of corresponding attributes are equal. Values of a tuple type's domain are essentially the same as tuples of the relational model with one difference. In the relational model, the domain of an attribute of a tuple is a basic type but an attribute's domain may be any semilattice data type in the semilattice model. The notation and operations of a tuple data type are described in Section 3.1.3.

The union type constructor is similar to the classification domain constructor of the Format model [HuY84] and the *case* statement in type definitions of the Pascal programming language. Like the tuple type constructor, the union type constructor forms a new type from a finite mapping of tags (names) to data types. An operation is defined for each pair in the mapping. The operation is a one-to-one function from the domain of the type of the pair into the domain of the union type. The union type's domain is the union of the disjoint sets produced by these operations. Two values of a union type are equal if and only if both were generated in the same manner. That is both their tags and underlying values are equal. Section 3.1.4 has more information about the union type constructor.

For the purposes of illustration, the type definitions of a semilattice database are given on the next page in Table 3.1. Values of these types can describe certain aspects of most scenes. Recall that in the semilattice model no constraints are given with a database schema.

Table 3.1 The Type Definitions of a Semilattice Database Schema

```
scene           = {object}
object          = (location:point,
                      description:(simple:simple-object|complex:composite-object))
simple-object   = (ball:(radius:real)|box:point|regular:{polygon}
                      |irregular:(from:plane,to:plane,
                          mapping:{(from:planar-point,to:planar-point)})
polygon         = (in:plane,vertices:<planar-point>)
plane           = (first:point,second:point,third:point)
point           = (x-co-ord:real,y-co-ord:real,z-co-ord:real)
planar-point    = (x-co-ord:real,y-co-ord:real)
composite-object = (components:<object>,operation:string)
```

Description

The type *scene* is a set type with values that are finite sets of the type *object*. The type *object* is a tuple with attributes for the position and description of the object. The type of the object's description is a union of the type *simple-object* and the type *composite-object*. The type *simple-object* is a union of other types of objects, *ball, box,* and so on, where each is described in terms of simpler types like *point, plane* and *polygon*. The type *composite-object* is a tuple with two attributes. One contains a sequence of objects and the other is a string naming the operation to be performed on the objects. When the attribute *operation* has the value *union*, the corresponding object is the set of points that is the union of the sets of points that make up the component objects.

3.1.1. The Set Type Constructor

The expression $\{d\}$ is a *set type expression* whenever d is a type expression. Values of this type are denoted by expressions of the form $\{v_1, v_2, \cdots, v_k\}$ where each v_i is the notation for a value of the type defined by d and $0 \leq k < \infty$. This notation allows many representations of the same set. Although a permutation of the order of the elements produces a different representation, the set described is the same. Also, the definition permits the notation for a set to have a repetition of the notation for an element. Although this may seem unreasonable, it is consistent with formal treatments of sets by others [MaW85, GHW85]. No problems arise from the notation if care is taken in specifying the operations of the data type.

The operations of a set data type are described below. The complete specification of these operations is in Sections 7 and 8 of the Larch Shared Language Handbook in [GHW85]. The symbol C denotes the type of the set data type. The type of the elements is denoted by E. All values of the data type are generated by the first two operations.

new : $\rightarrow C$

This operation returns the empty set.

insert : $C, E \rightarrow C$

This operation returns a set with the new element added to the given set.

Note: C, E is the Larch notation for the cartesian product $C \times E$.

isEmpty : $C \rightarrow$ *boolean*

This operation returns *true* if the set is empty and *false* otherwise.

size : $C \rightarrow$ *integer*

This operation returns a count of the number of elements in the set.

$\# \in \#$: E, $C \rightarrow$ *boolean*

This operation returns *true* if the given element is in the given set and *false* otherwise.

$\# = \#$: C, $C \rightarrow$ *boolean*

This operation returns *true* if the two given sets are equal and *false* otherwise. Two sets c_1 and c_2 are equal if and only if $\forall x \in E$, $x \in c_1$ if and only if $x \in c_2$.

delete : C, $E \rightarrow C$

This operation returns a set with the given element deleted from the given set. If the element is not in the set then the given set is returned.

next : $C \rightarrow E$

This operation returns some element of a non-empty set. This operation is not defined for the empty set.

rest : $C \rightarrow E$

This operation, given a non-empty set, returns the set produced by deleting from the set the element returned by the *next* operation on the set. This operation is not defined for the empty set.

$\# \cup \#$: C, $C \rightarrow C$

This operation returns the union of the two given sets.

$\# \cap \#$: C, $C \rightarrow C$

This operation returns the intersection of the two given sets.

$\#\subseteq\#$: *C, C→boolean*

This operation returns *true* if the first set is a subset of the second set and *false* otherwise. The empty set is a subset of all sets. A non-empty set c_1 is a subset of a second set c_2 if and only if $next(c_1) \in c_2$ and $rest(c_1) \subseteq c_2$.

$\#\subset\#$: *C, C→boolean*

This operation returns *true* if the first set is a proper subset of the second set and *false* otherwise. The first set is a proper subset of the second set if and only if it a subset of the second set but not equal to the second set.

3.1.2. The Sequence Type Constructor

The expression $<d>$ is a *sequence type expression* whenever d is a type expression. Values of this type are denoted by expressions of the form $<v_1, v_2, \cdots, v_k>$ where each v_i is the notation for a value of the type defined by d and $0 \leq k < \infty$. This notation has only one representation of each sequence whenever the elements are of a type that has only one representation for each element.

The operations of a sequence data type are described below. The complete specification of these operations is in Sections 7 and 8 of the Larch Shared Language Handbook in [GHW85]. The symbol C denotes the type of the sequence data type. The type of the elements is denoted by E. All values of the data type are generated by the first two operations.

new : →*C*

This operation returns the empty sequence.

insert : *C, E→C*

This operation returns a sequence with the given element added to the front of the given sequence.

enter : $C, E \rightarrow C$

This operation returns a sequence with the given element added to the end of the given sequence.

isEmpty : $C \rightarrow boolean$

This operation returns *true* if the sequence is empty and *false* otherwise.

size : $C \rightarrow integer$

This operation returns the number of elements in the given sequence.

count : $C, E \rightarrow integer$

This operation returns a count of the number of times that the given element appears in the given sequence.

$\# = \#$: $C, C \rightarrow boolean$

This operation returns *true* if the two given sequences are equal and *false* otherwise. A non-empty sequence is never equal to the empty sequence. Two non-empty sequences c_1 and c_2 are equal if and only if $first(c_1) = first(c_2)$ and $rest(c_1) = rest(c_2)$.

first : $C \rightarrow E$

This operation returns the first element of a non-empty sequence. This operation is not defined for the empty sequence.

rest : $C \rightarrow C$

This operation returns a sequence that has all but the first element of a given non-empty sequence. This operation is not defined for the empty sequence.

last : $C \rightarrow E$

This operation returns the last element of a non-empty sequence. This operation is not defined for the empty sequence.

prefix : $C \rightarrow C$

This operation returns a sequence that has all but the last element of a given non-empty sequence. This operation is not defined for the empty sequence.

$\#\,||\,\# : C, C \rightarrow C$

This operation returns the sequence that is the concatenation of the two given sequences.

$\#[\#] : C, integer \rightarrow E$

This operation, given a sequence c and an integer i, returns the ith element of the sequence if $size(c) \geq i \geq 1$. This operation is not defined otherwise.

3.1.3. The Tuple Type Constructor

The expression $(a_1:d_1, a_2:d_2, \cdots, a_k:d_k)$ is a *tuple type expression* if each a_i is a distinct name and d_i is a type expression for all $i \in \{1,2, \cdots, k\}$ for some k, $1 \leq k < \infty$. The order of the pairs of an attribute name and a type expression is not significant in any way. Therefore any tuple data type with k attributes is defined by $k!$ different tuple type expressions. Values of a tuple type are denoted by expressions of the form $(a_1 = v_1, a_2 = v_2, \cdots, a_k = v_k)$ where v_i is the notation for a value in type defined by d_i for each $i \in \{1,2, \cdots, k\}$. Here there are at least $k!$ equivalent representation of each value.

The operations of a tuple data type are described below. The symbol T denotes the type of the tuple data type. The type for the expression d_i is denoted by T_i.

$(a_1 = \#, a_2 = \#, \cdots, a_k = \#) : T_1, T_2, \cdots, T_k \rightarrow T$

This operation returns the tuple for the given values. All values of the tuple data type are generated by this operation.

$\#.a_1 : T \rightarrow T_1$

$\#.a_2 : T \rightarrow T_2$

$\cdots$

$\#.a_k : T \rightarrow T_k$

These operations return the value of the appropriate attribute for a given tuple.

$\# = \# : T, T \rightarrow boolean$

This operation returns *true* if the two given tuples are equal and *false* otherwise. Two tuples t and t' are equal if and only if $\forall\, a_i \in \{a_1, a_2, \cdots, a_k\}\ (t.a_i = t'.a_i)$.

3.1.4. The Union Type Constructor

The expression $(t_1{:}d_1 | t_2{:}d_2 | \cdots | t_k{:}d_k)$ is a *union type expression* if each t_i is a distinct name and d_i is a type expression for all $i \in \{1,2,\cdots,k\}$ for some k, $2 \leq k < \infty$. Each pair of a tag name and a type expression is called a domain of the union. The order of the domains is not significant in any way. Therefore any union data type with k domains is defined by $k!$ different union type expressions. Values of a union type are denoted by expressions of the form (tag$=t$,value$=v$) where $t = t_i$ and v is the notation for a value of the type defined by d_i for some $i \in \{1,2,\cdots,k\}$. There is one representation for a value in the union domain for each representation of a value in the subordinate domains.

The operations of a union data type are described below. The symbol T denotes the type of the union data type. The type for the expression d_i is denoted by T_i.

(tag= t_1,value= #) : $T_1 \rightarrow T$

(tag= t_2,value= #) : $T_2 \rightarrow T$

. . .

(tag= t_k,value= #) : $T_k \rightarrow T$

These operations each return a union value. All values of the data type are generated by the these operations.

#.tag : $T \rightarrow$ name

This operation returns the tag of a given value of the union type.

#.t_1 : $T \rightarrow T_1$

#.t_2 : $T \rightarrow T_2$

. . .

#.t_k : $T \rightarrow T_k$

These operations return the value of a given union value if the tag of that value is the name of the operation. The operation #.t_i is not defined for any union value with a tag other than t_i.

= # : T, $T \rightarrow$ *boolean*

This operation returns *true* if the two given union values are equal and *false* otherwise. Two union values (tag= t,value= v) and (tag= t',value= v') are equal if and only if $t = t'$ and $v = v'$.

3.1.5. Definitions

A *type definition* consists of a type name and a type expression separated by an equal sign: *name = type expression*. The type expression is the name of a type or it is formed from other type expressions by one of the four type constructors. Note that recursive definition of types is permitted.

The *domain* of a semilattice data type is the set of all data values produced by the generating operations of the data type. The notation used to denote the values of basic types is the same as a conventional programming language. For constructed types the notation is described in the preceding sections. Recall that all values must be finite even though it is possible to construe the definitions as allowing values that are infinite recursions.

A *database schema* is a set of semilattice type definitions that is consistent and complete. A set of definitions is consistent if each definition has a unique name. A set of definitions is complete if it has a definition for any type whose name appears in any of the type definitions. The notation $dom[S,n]$[1] identifies the domain of the type named n in the database schema S. The domain of the database schema S is denoted by $Dom[S]$. It is the union of the domains of all types in the schema. The *reference graph* of a database schema has a node for each type definition and an arc (n_1,n_2) if n_2 is a sub-expression of the expression that defines n_1.

A type n *references* a type n' if there is a path $n,n_1,n_2,\cdots,n_i,n'$ $(i\geq 0)$ in the reference graph. A type n is *recursive* if it references itself. A type is *simply recursive* if it is self-referencing independent of its set and sequence type constructors. Formally, a type is simply recursive if the name of the type is a sub-expression in the simple expression associated with the type. The *simple expression* associated with a type is the simplification of the expression that defines the type. The *simplification* of a type expression is the expression with all set and sequence sub-expressions replaced by the expression *singleton*.

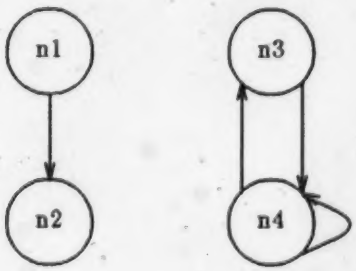
---

[1] In this thesis, the parameters of functions are placed inside square brackets. This makes the specification of functions on tuple and union values easier to read. Otherwise, the equations would contain more parentheses and it would be more difficult to match corresponding ones.

An example of a semilattice database schema, called $S1$, is

$$\{n1=(a{:}real, b{:}string, c{:}n2), n2=(f{:}real, g{:}integer),$$
$$n3=\{(a{:}n4, b{:}boolean, c{:}<\{string\}>)\},$$
$$n4=(t1{:}n4 \mid t2{:}n3 \mid t3{:}<(a{:}real, b{:}real)>)\}.$$

The set $\{(a=v_a, b=v_b, c=(f=v_f, g=v_g)) \mid v_a, v_f \in real, v_b \in string$ and $v_g \in integer\}$ contains one representation of each element in $dom[S1, n1]$, the domain of the type $n1$ of the schema $S1$. In this database schema, the type $n3$ is recursive but not simply recursive. The type $n4$ is simply recursive since it is in the simple expression $(t1{:}n4 \mid t2{:}n3 \mid t3{:}singleton)$ that is associated with the type. The reference graph for this database is shown below.



3.2. A Comparison of the Semilattice and Relational Data Model

The semilattice data model is intended to avoid the perceived deficiencies of the relational model of E. F. Codd [Cod70]. This is to be done without sacrificing the relational model's ability to handle data in an abstract and eloquent manner. The semilattice data model, like the relational model, has mathematical eloquence, machine independence and a method of representing common data that effectively manages redundancy. It is felt that the semilattice data model makes efficient use of secondary storage. Efficiency is achieved by several techniques including using the mathematical concept of a semilattice to efficiently store shared data. Information about the physical implementation of the semilattice data model is available in Ajit Singh's thesis

[Sin86].

The sharing of common data allows the semilattice data model to have the advantages of the relational data model with the performance of the network data model. Some complaints about the relational model have focused on performance problems of implementations. A relational query may require several join operations. Depending of the data and the implementation each of these joins may require a great deal of processing. Some queries require join operations to combine related data that was separated to produce a schema in the desired normal form. This decomposition is contrary to the database design philosophy of the semilattice data model. Therefore, the join operation will seldom be used in database queries.

Other criticisms have been made of the relational model's requirement that data be in first normal form. As well as losing potential performance benefits, this requirement makes it extremely difficult to describe data that is heterogeneous in nature. It has been pointed out that:

> "While solutions to these problems are possible, they are contrary to the reason for employing a database management system in the first place: to handle the design information in a clean, consistent fashion, rather than employ ad hoc techniques." [HNC84]

> "Hierarchies and heterogeneous relations can be simulated in the relational model. . . . However simulated relations are more cumbersome than concrete ones." [Har84]

These criticisms can not be made of the semilattice data model because it allows type definitions in database schemas that are able to handle these situations with ease.

In the semilattice data model, a database schema is a set of schemas that each define a data type. The relational data model can also be viewed as having database schemas that are set of schemas the each define a data type. If this is done, a major difference between the two data models is apparent. Each relational schema's set of types is a set of relations. Using semilattice terms, a relation is a type constructed by

applying the set constructor to any tuple type constructed from basic types. Clearly, these are some of the simplest database schemas of the semilattice data model. In contrast, schemas of the semilattice data model have an infinite variety of styles. It may be asked if the more complex types serve any practical value. In many situations, they do since they avoid the need to design the conceptual schema to accommodate the restrictions of the relational model. For instance, at each step in the fabrication of an integrated circuit, a set of polygons describes the area to process. A typical normalized relational description of a set of polygons is a relation where each tuple contains five attributes: a polygon identification number, and the x and y co-ordinates of both end points of a line forming one side of the polygon. A semilattice description of a set of polygons is a set where each element describes a polygon by a sequence of the vertices. The semilattice representation requires less storage than the normalized relational description since it only stores the position of each vertex once and does not require artificial keys like the polygon identification number. Furthermore, the semilattice representation assures data of a certain quality since every sequence of vertices is a polygon but most sets of lines are not polygons.

Another large contrast between the two data models is seen when their need for null values is compared. The normalized relational model uses null values when no appropriate value exists in an attribute's domain. This happens because the value does not exist, is unknown, is being changed, is confidential or is unavailable for some other reason. A complex theory has evolved that circumvents the problems created by using null values. It is intended that the semilattice model have no null values. It is believed that the model is flexible enough to handle those situations that typically require nulls in the relational model. The semilattice type constructors can describe data types capable of directly holding data for those situations. In particular, any situation requiring the empty set as a value is easily handled by the set type constructor. The union type constructor allows any number of optional fields to belong to a

type. For example, consider a field that typically holds an integer and has several different nulls that are used to indicate different reason why no value is present. In the semilattice data model, the expression

$(normal{:}integer \mid reason_1{:}singleton \mid reason_2{:}singleton \mid \cdots \mid reason_x{:}singleton)$

could be used to define the domain of the field.

3.3. Semilattice Schema Domains

In this section, some properties of the domains of semilattice schemas are given. These results are based on the use of context-free grammars that generated a set of strings that contains one or more representation of every value in a schema's domain. It is assumed that the set of strings used to denote the values of each basic type of the semilattice model is a regular set. This assumption is valid for all proposed basic domains. The first result shows that although the recursively defined data type of the semilattice data model have rich domains, context-free grammars are adequate to describe their notation.

Theorem 3.1

For any database schema S, there is a context-free grammar G such that

$\forall \omega \in L[G]$ there exists an $x \in Dom[S]$ such that ω is a notation for x and

$\forall x \in Dom[S]$ there exists an $\omega \in L[G]$ such that ω is a notation for x.

Proof (by construction)

A method of constructing a grammar from any database schema follows.

The grammar's sentence symbol is Σ. All of the non-terminals are of the form $n.i.j$. The non-terminal $n.i.j$ generates the notation for the ith subexpression of the jth subexpression of the type n.

The grammar's set of rules is the union of the set of rules for generating values of the basic type, a set with a rule $\Sigma \rightarrow n$ for each named type n and a set of rules for each

type definition. The function *rules* determines the set of rules for each type definition from the type name and the associated type expression. A specification of this function is given below. Since the comma symbol and set braces, {}, are terminals of the grammar, the specification of the function denotes a set of rules $\{r_1, r_2, \cdots, r_i\}$ as $\lceil r_1; r_2; \cdots; r_i \rceil$.

$$rules[n, n'] = \lceil n \rightarrow n' \rceil$$

$$rules[n, \{d\}] = \lceil n \rightarrow \{\}\ ;\quad n \rightarrow \{\underline{n.1}\}\ ;\quad \underline{n.1} \rightarrow \underline{n.1}, \underline{n.1} \rceil \cup rules[n.1, d]$$

$$rules[n, <d>] = \lceil n \rightarrow <>\ ;\ n \rightarrow <\underline{n.1}>\ ;\quad \underline{n.1} \rightarrow \underline{n.1}, \underline{n.1} \rceil \cup rules[n.1, d]$$

$$rules[n, (a_1 : d_1, a_2 : d_2, \cdots, a_k : d_k)] = \lceil n \rightarrow (a_1 = \underline{n.1}, a_2 = \underline{n.2}, \cdots, a_k = \underline{n.k}) \rceil \bigcup_{i=1}^{k} rules[n.i, d_i]$$

$$rules[n, (t_1 : d_1 | t_2 : d_2 | \cdots | t_k : d_k)] = \left\lceil \begin{array}{l} n \rightarrow (\text{tag} = t_1, \text{value} = \underline{n.1}); \\ n \rightarrow (\text{tag} = t_2, \text{value} = \underline{n.2}); \cdots; \\ n \rightarrow (\text{tag} = t_k, \text{value} = \underline{n.k}) \end{array} \right\rceil \bigcup_{i=1}^{k} rules[n.i, d_i]$$

Example

The database schema *S*1 of Section 3.1.5 is used for illustration. In addition to the rules for generating the basic types, a grammar derived for this schema has the rules given below.

$\Sigma \rightarrow \underline{n1}$ $\underline{n1} \rightarrow (a = \underline{n1.1}, b = \underline{n1.2}, c = \underline{n1.3})$ $\underline{n1.1} \rightarrow \underline{real}$ $\underline{n1.2} \rightarrow \underline{string}$ $\underline{n1.3} \rightarrow \underline{n2}$

$\Sigma \rightarrow \underline{n2}$ $\underline{n2} \rightarrow (f = \underline{n2.1}, g = \underline{n2.2})$ $\underline{n2.1} \rightarrow \underline{real}$ $\underline{n2.2} \rightarrow \underline{integer}$

$\Sigma \rightarrow \underline{n3}$ $\underline{n3} \rightarrow \{\}$ $\underline{n3} \rightarrow \{\underline{n3.1}\}$ $\underline{n3.1} \rightarrow \underline{n3.1}, \underline{n3.1}$
$\underline{n3.1} \rightarrow (a = \underline{n3.1.1}, b = \underline{n3.1.2}, c = \underline{n3.1.3})$ $\underline{n3.1.1} \rightarrow \underline{n4}$ $\underline{n3.1.2} \rightarrow \underline{boolean}$
$\underline{n3.1.3} \rightarrow <>$ $\underline{n3.1.3} \rightarrow <\underline{n3.1.3.1}>$ $\underline{n3.1.3.1} \rightarrow \underline{n3.1.3.1}, \underline{n3.1.3.1}$
$\underline{n3.1.3.1} \rightarrow \{\}$ $\underline{n3.1.3.1} \rightarrow \{\underline{n3.1.3.1.1}\}$
$\underline{n3.1.3.1.1} \rightarrow \underline{n3.1.3.1.1}, \underline{n3.1.3.1.1}$ $\underline{n3.1.3.1.1} \rightarrow \underline{string}$

$\Sigma \rightarrow \underline{n4}$ · $\underline{n4} \rightarrow (\text{tag} = t1, \text{value} = \underline{n4.1})$ $\underline{n4.1} \rightarrow \underline{n4}$
$\underline{n4} \rightarrow (\text{tag} = t2, \text{value} = \underline{n4.2})$ $\underline{n4.2} \rightarrow \underline{n3}$
$\underline{n4} \rightarrow (\text{tag} = t3, \text{value} = \underline{n4.3})$ $\underline{n4.3} \rightarrow <>$ $\underline{n4.3} \rightarrow <\underline{n4.3.1}>$
$\underline{n4.3.1} \rightarrow \underline{n4.3.1}, \underline{n4.3.1}$ $\underline{n4.3.1} \rightarrow (a = \underline{n4.3.1.1}, b = \underline{n4.3.1.2})$
$\underline{n4.3.1.1} \rightarrow \textit{real}$ $\underline{n4.3.1.2} \rightarrow \textit{real}$

An important corollary to Theorem 3.1 is essential to the credibility of the information-preserving transformations described in Section 4.4. These corrective transformations only apply when a schema has a type with the empty domain. The types are called *valueless* in the remaider of this thesis. Although the schema $\{n = (a{:}n)\}$ is well-formed syntactically, there is no way to create a tuple of the type n since there is no way to create the first tuple of that type. The corollary states that valueless types are always detectable.

Corollary

For any data type n of any semilattice database schema S, it is decidable if $dom[S,n] = \varnothing$.

Proof

The language of the non-terminal $\underline{n}$ of the grammar of the previous theorem is empty if and only if $dom[S,n] = \varnothing$. From Theorem 9.1 of [DDQ78], it is decidable if the language generated by a non-terminal of a context-free grammar is empty.

Given regular grammars that generate the notation for the basic domains, the closure rules of regular sets guarantee that there is a regular grammar related to the domain of any semilattice schema without recursive types. Nevertheless, it is reasonable to use context-free grammars to describe the domains of database schemas

because regular grammars are not adequate for some schemas with recursive types. In fact, the schemas with recursive types that can be described by regular grammars are exactly those that have no recursive types when each valueless type is replaced by any type not referring to other domains.

The next result shows that there exists some semilattice schemas whose domains are not regular. The proof requires the UVW theorem. This is called the pumping lemma for regular sets by Hopcroft and Ullman in [HoU79]. Their statement of the theorem is:

> **Lemma 3.1** - Let L be a regular set. Then there is a constant n such that if z is any word in L, and $|z| \geq n$, we may write $z = uvw$ in such a way that $|uv| \leq n$, $|v| \geq 1$, and for all $i \geq 0$, $uv^i w$ is in L.

Theorem 3.2

There is a database schema whose notation is not regular.

Proof (by example)

Consider the database schema $S = \{n = (t1:(a:n, b:real) | t2:string)\}$. The notation for S is the set

$$N_S = \{ (\text{tag} = t1, \text{value} = (a = (\text{tag} = t1, \text{value} = (a = (\text{tag} = t1, \text{value} = \cdots$$
$$(\text{tag} = t1, \text{value} = (a = (\text{tag} = t2, \text{value} = v_{k+1}), b = v_k) \cdots), b = v_2), b = v_1)$$
$$\mid k \geq 0,\ v_1, v_2, \cdots, v_k \in real,\ v_{k+1} \in string\}$$

Assume that N_S is regular. In that case, the pumping lemma of regular sets applies. Apply the lemma to some large string in N_S to generate a string that is not in N_S because it has an improper form (mismatched parenthesis).

Conclude that N_S is not regular.

Chapter 4

Information-preserving Transformations

This chapter discusses information-preserving transformations of database schemas of the semilattice data model. It defines the term information-preserving transformation and describes those transformations that have been studied. Since there are an infinite number of information-preserving transformations, only a few transformations are presented. The intention is to omit any transformation that can be decomposed into a sequence of simplier transformations. Transformations that appear to manipulate data in an inappropriate manner are also excluded.

The chapter starts with a section that defines the terms informational equivalence and information-preserving transformation. The first section then elaborates on the criteria used to select the transformation that are presented. In the second section of the chapter, some non-structural transformations are briefly covered. Next, some structure-changing transformations are documented. This is followed by the descriptions of the corrective transformations. The corrective transformation eliminate the portions of a semilattice database schema that do not generate values. The last section explains why this chapter includes only part of Codd's normalization transformation. That section also describes another schema transformation that was originally included in this chapter. Although the transformation is information preserving, it was excluded because it uses inappropriate methods of transforming data values.

4.1. Definitions and Discussion

To define the term information-preserving transformation, a definition of informational equivalence for schemas is required. The definition is similar in style to some of the definitions mentioned in Chapter 2 [Bor80, AAB82, Hul84].

Database schemas S and S' are *informationally equivalent* (written $S \equiv_I S'$) if there exists two computable functions $f{:}Dom[S] \rightarrow Dom[S']$ and $g{:}Dom[S'] \rightarrow Dom[S]$

such that $\forall x \in Dom[S](g[f[x]] = x)$ and $\forall x \in Dom[S'](f[g[x]] = x)$. The two functions show that the schemas are equivalent. Hereafter, the functions are called *data transformation functions*. An equivalence relation must have the properties of symmetry, reflexivity and transitivity. Symmetry follows from the symmetry of the definition. Using the identity function for both data transformation functions gives reflexivity. The composition of functions gives transitivity.

A *schema transformation* is any binary relation on database schemas. By definition, a schema transformation T is *information preserving* if $\forall (S,S') \in T$, $S \equiv_I S'$. The definition's symmetry guarantees that the inverse of an information-preserving transformation is also information preserving. Since $\equiv_I$ is an equivalence relation, the composition of information-preserving transformations is also information preserving.

By using the identity function for both of the data transformation functions, it can be shown that the following two schemas are equivalent.

$$\{s1 = (a{:}real, b{:}string, c{:}s2), s2 = (f{:}real, g{:}integer)\}$$
$$\{s1 = (a{:}real, b{:}string, c{:}(f{:}real, g{:}integer)), s2 = (f{:}real, g{:}integer)\}$$

This pair of schemas is an element of the transformation *substitute a type's definition for its name*. This transformation is described in Section 4.2.1.

The remainder of this section discusses the definition of equivalence and how transformations were selected for this chapter.

For the semilattice data model, Hull and Yap's definition of equivalence is unsatisfactory. With their definition, schemas are equivalent if their domain's cardinality is the same for all cardinalities of the basic domains. This definition was rejected for two reasons. Requiring equivalent schemas to have the same domain size when the cardinality of the boolean domain is either one or a thousand is unreasonable. More importantly, the definition treats many unrelated semilattice database schemas as equivalent since the schemas always have a countable infinite number of possible

databases unless most of the basic domains are empty. For instance, the schemas $\{n=(a{:}real, b{:}<real>)\}$, $\{n=<real>\}$, $\{n=\{<real>\}\}$, and $\{n=(a{:}real|b{:}n)\}$ would all be considered equivalent. Clearly, the definition requires some fundamental changes to handle sequences and recursion.

Instead of reworking Hull and Yap's definition, the style of other definitions mentioned in Chapter 2 [Bor80, AAB82, Hul84] was adopted. These definitions require a one-to-one function the domain of one schema onto the domain the other schema. One value of this the function is that it provides a mechanism to transform any database of one schemas to a database of the other schema. Furthermore, the absence of some function with the given properties is unacceptable since then a database must exists for one of the schemas which is not represented by any database of the other schema.

The given definition of informational equivalence has a deficiency. The definition permits a data transformation function to be any computable function. Some computable functions perform undesired manipulation of data values. For instance, some functions encode strings as integers and this is not desired. Another problem with allowing any computable function is that every database schema is equivalent to some schema with a single, simple type. From the results of Section 3.3, it follows immediately that all schemas have domains with a finite or a countably infinite number of elements. By using functions based on *Gödel* numbering, all database schemas with an infinite number of valid databases are equivalent to the schema $\{n=integer\}$. The schema $\{n=(t1{:}singleton|t2{:}singleton|\cdots|tn{:}singleton)\}$ is equivalent to any database schema with n valid databases by the obvious function. Efficiency is another problem resulting from allowing a data transformation function to be any computable function. Most computable functions are intractable. In fact, most tractable functions are infeasible for large databases.

The definition of equivalence in [AAB82] is based on queries. It requires the data

transformation functions used to show equivalence to be a group of queries from the query language of the data model. This restriction may seem too demanding but it has two large benefits. It guarantees that the data transformation functions are computable in a practical way and it fulfills Borkin's requirement [Bor80] that equivalent schemas must be able to process each other's operations.

This thesis should use the query-based definition of equivalence of [AAB82]. Unfortunately, the query language of the semilattice data model is still in development. The design of a database query language is a difficult task. For the semilattice data model, the design is more difficult because of the diverse and recursive nature of schema definitions. Since, at this time, a query-based definition of equivalence can not be used, the given definition of equivalence is used. Nevertheless, the spirit of a query-based definition is followed in this thesis. A subjective judgement was made about the nature of the data transformation functions of each information-preserving transformation considered for inclusion in this chapter. The only transformations allowed were those that have simple data transformation functions that are query-like in style. The assessment of a function's query-likeness was based on two commonly referenced principles of data retrieval languages:

> (1) that the value produced by a query should be independent of the manner in which the data are actually stored in the database and
>
> (2) that a query language should treat data values as essentially uninterpreted objects, although certain properties, such as a linear ordering on certain domains can be built into the query language. [AhU79].

4.2. Non-structural Information-preserving Transformations

Some information-preserving transformations do not change the structure of the data values. In fact, since distinct schemas may have the same domain, some transformations do not change the data values at all. Although these transformations may seem to be trivial, some are useful in simplifying a database schema. Whether trivial or not, the transformations are documented here for completeness.

4.2.1. Substitute a Type's Definition for Its Name

When the name of a constructed type is a sub-expression of a type definition, replace the type name by the expression that defines it. Values remain unchanged.

4.2.2. Replace Two Identical Types by One Type

If a schema has two type definitions that are identical except for name, this transformation produces a schema with the two definitions replaced by one. Given a schema with two type definitions $n_1 = d_1$ and $n_2 = d_2$ where d_1 and d_2 are equivalent type expressions, then remove the type definition for n_2 and replace each reference to it by a reference to n_1. Two type expressions are equivalent if one is produced from the other by some series of permutations of its tuple and union sub-expressions.

4.2.3. Change a Type's Name

Any type name may be changed to any other name provided that the new name is not the name of another type in the database schema. Any type definitions having the old name are changed to have the new name.

4.2.4. Change a Tag or Attribute Name

Any attribute name in a type definition may be changed to another name provided that the new name is not the same as another attribute belonging to the same tuple constructor. With similar restrictions, any tag in a union may be changed. Any value of the tuple or union requires a straightforward update.

4.3. Structure-changing Information-preserving Transformations

This section describes some information-preserving transformations that might be made to a reasonably designed database. Each description starts with an informal description and an example of the transformation. Then a formal definition and a proof that the transformation is information preserving is provided. Remarks about problems or features are included for some transformations.

For clarity, certain statements are omitted from the formal definitions and the reader should assume the following points. The transformation is from a database schema S to another schema S'. In a few transformations, an entire type definition is replaced or removed. The formal definition is explicit in those cases. Otherwise, the difference between the two schemas is that some sub-expression in one of the type definitions is replaced by a different expression. In the definitions, any symbol like d_i denotes some type expression. Any restrictions on these expressions are explicitly stated.

The description of each transformation has a proof of information preservation. The proof is a specification of the generic functions f and g. These show how to construct the functions f_S and g_S for a particular S. Since $\forall x \in Dom[S](g_S[f_S[x]] = x)$ and $\forall y \in Dom[S'](f_S[g_S[y]] = y)$, it follows that $S \equiv_I S'$. For brevity, the specifications of f and g are incomplete. Only the specifications for the types of the transformed expression are given. The specification of f_S can be completed by adding an equation for each other type expression. The specification of g_S can be completed in the same manner. The form of the equation added for a type is the appropriate one from the following list.

$f_S[(a_1=v_1, a_2=v_2, \cdots, a_k=v_k)]$	$=(a_1=f_S[v_1], a_2=f_S[v_2], \cdots, a_k=f_S[v_k])$
$f_S[(\text{tag}=t, \text{value}=v)]$	$=(\text{tag}=t, \text{value}=f_S[v])$
$f_S[\{v_1, v_2, \cdots, v_k\}]$	$=\{f_S[v_1], f_S[v_2], \cdots, f_S[v_k]\}$
$f_S[<v_1, v_2, \cdots, v_k>]$	$=<f_S[v_1], f_S[v_2], \cdots, f_S[v_k]>$
$f_S[x]$	$=x$ (x is an element of a basic type)

4.3.1. Include a Subordinate Tuple's Attributes

A tuple with an attribute whose domain is a tuple is transformed to a tuple with that attribute replaced by the attributes of the subordinate tuple. The two sets of names of attributes combined by this transformation may have common elements. To avoid problems, each attribute is renamed appropriately. An example of this transformation is replacing the expression (a:*real*,b:*string*,c:(f:*real*,g:*integer*)) by the expression (*sup_a*:*real*,*sup_b*:*string*,*sub_f*:*real*,*sub_g*:*integer*). The basis for this transformation is Codd's normalization [Cod70] and Hull and Yap's *comp-comp* reduction [HuY84].

Formal Definition

Given an expression $(a_1:d_1, a_2:d_2, \cdots, a_k:d_k)$

where for some i, $1 \le i \le k$, d_i is $(a_{i_1}:d_{i_1}, a_{i_2}:d_{i_2}, \cdots, a_{i_m}:d_{i_m})$

and assuming without loss of generality that $i=k$,

replace the expression by $(\overline{a}_1{:}d_1, \overline{a}_2{:}d_2, \cdots, \overline{a}_{k-1}{:}d_{k-1}, \overline{a}_{k_1}{:}d_{k_1}, \overline{a}_{k_2}{:}d_{k_2}, \cdots, \overline{a}_{k_m}{:}d_{k_m})$.

To avoid a tuple schema with duplicate attribute names, a function renames each attribute name of the subordinate tuple and another function renames each attribute name of the superordinate tuple. The first function concatenates the string *sub_* and the given attribute name. The other function concatenates the string *sup_*. and the given attribute name. The notation $\overline{a}_j$ denotes the result of the appropriate function on a_j.

Proof of information preservation

Specify the generic functions used to construct the data transformation functions.

$$f[(a_1 = v_1, a_2 = v_2, \cdots, a_{k-1} = v_{k-1}, a_k = (a_{k_1} = v_{k_1}, a_{k_2} = v_{k_2}, \cdots, a_{k_m} = v_{k_m}))]$$
$$= (\overline{a}_1 = f[v_1], \overline{a}_2 = f[v_2], \cdots, \overline{a}_{k-1} = f[v_{k-1}], \overline{a}_{k_1} = f[v_{k_1}], \overline{a}_{k_2} = f[v_{k_2}], \cdots, \overline{a}_{k_m} = f[v_{k_m}])$$

$$g[(\overline{a}_1 = v_1, \overline{a}_2 = v_2, \cdots, \overline{a}_{k-1} = v_{k-1}, \overline{a}_{k_1} = v_{i_1}, \overline{a}_{k_2} = v_{k_2}, \cdots, \overline{a}_{k_m} = v_{k_m})]$$
$$= (a_1 = g[v_1], a_2 = g[v_2], \cdots, a_{k-1} = g[v_{k-1}], a_k = (a_{k_1} = g[v_{k_1}], a_{k_2} = g[v_{i_2}], \cdots, a_{k_m} = g[v_{k_m}]))$$

Remarks

The inverse transformation creates a subordinate tuple from a group of attributes. This would be useful in sharing data between two types with common attributes. Unfortunately, the inverse transformation is not appropriate for a Church-Rosser replacement system.

4.3.2. Include a Subordinate Union's Domains

Transform a union with a domain that is also a union by replacing that domain by all of the domains of the subordinate union. The two sets of domain tags combined by this transformation may have common elements. To avoid problems, each tag is renamed appropriately. For example, this transformation replaces the expression $(a{:}real|b{:}string|c{:}(f{:}real|g{:}integer))$ by the equivalent expression

$$(sup_a{:}real|sup_b{:}string|sub_f{:}real|sub_g{:}integer).$$

This transformation is based on Hull and Yap's *class-class* reduction [HuY84].

Formal Definition

Given an expression $(t_1{:}d_1|t_2{:}d_2|\cdots|t_k{:}d_k)$

where for some i, $1 \le i \le k$, d_i is $(t_{i_1}{:}d_{i_1}|t_{i_2}{:}d_{i_2}|\cdots|t_{i_m}{:}d_{i_m})$,

replace the expression by

$$(\overline{t}_1{:}d_1|\overline{t}_2{:}d_2|\cdots|\overline{t}_{i-1}{:}d_{i-1}|\overline{t}_{i_1}{:}d_{i_1}|\overline{t}_{i_2}{:}d_{i_2}|\cdots|\overline{t}_{i_m}{:}d_{i_m}|\overline{t}_{i+1}{:}d_{i+1}|\cdots|\overline{t}_k{:}d_k).$$

To avoid a union schema with duplicate tag names, a function renames each tag of the subordinate union and another function renames each tag of the superordinate union. The first function concatenates the string *sub_* and the given tag The other function concatenates the string *sup_*. and the given tag. The notation $\overline{t}_j$ denotes the result of the appropriate function on t_j.

Proof of information preservation

Specify the generic functions used to construct the data transformation functions.

$f[(\text{tag}=t,\text{value}=v)] = \text{if } t=\overline{t}_i \text{ then } \overline{f}[v] \text{ else } (\text{tag}=\overline{t},\text{value}=f[v])$
$\overline{f}[(\text{tag}=t,\text{value}=v)] = (\text{tag}=\overline{t},\text{value}=f[v])$

$g[(\text{tag}=\overline{t},\text{value}=v)]=$ if $\overline{t}\in\{\overline{t_{i_1}},\overline{t_{i_2}},\cdots,\overline{t_{i_m}}\}$ then $(\text{tag}=t_i,\text{value}=(\text{tag}=t,\text{value}=g[v]))$
else $(\text{tag}=t,\text{value}=g[v])$

4.3.3. Change a Tuple with a Union to a Union of Tuples

A tuple containing an attribute whose domain is a union is transformed to a union with a tuple made for each domain of the union. This is done by modifying the original tuple to have the union's domain as the domain of the attribute with the union. For example, replace the expression $(a{:}real,b{:}string,c{:}(f{:}real|g{:}integer))$ by $(f{:}(a{:}real,b{:}string,c{:}real)|g{:}(a{:}real,b{:}string,c{:}integer))$. This transformation is based on Hull and Yap's *comp-class* reduction [HuY84].

Formal Definition

Given an expression $(a_1{:}d_1,a_2{:}d_2,\cdots,a_k{:}d_k)$

where for some i, $1\le i\le k$, d_i is $(t_{i_1}{:}d_{i_1}|t_{i_2}{:}d_{i_2}|\cdots|t_{i_m}{:}d_{i_m})$,

replace the expression by

$$(t_{i_1}{:}(a_1{:}d_1,a_2{:}d_2,\cdots,a_{i-1}{:}d_{i-1},a_i{:}d_{i_1},a_{i+1}{:}d_{i+1},\cdots,a_k{:}d_k)$$
$$|t_{i_2}{:}(a_1{:}d_1,a_2{:}d_2,\cdots,a_{i-1}{:}d_{i-1},a_i{:}d_{i_2},a_{i+1}{:}d_{i+1},\cdots,a_k{:}d_k)$$
$$|\cdots$$
$$|t_{i_k}{:}(a_1{:}d_1,a_2{:}d_2,\cdots,a_{i-1}{:}d_{i-1},a_i{:}d_{i_k},a_{i+1}{:}d_{i+1},\cdots,a_k{:}d_k)).$$

Proof of information preservation

Specify the generic functions used to construct the data transformation functions.

$$f[(a_1=v_1,a_2=v_2,\cdots,a_{i-1}=v_{i-1},a_i=(\text{tag}=t,\text{value}=v),a_{i+1}=v_{i+1},\cdots,a_k=v_k)]$$
$$=(\text{tag}=t,\text{value}=(a_1=f[v_1],a_2=f[v_2],\cdots,$$
$$a_{i-1}=f[v_{i-1}],a_i=f[v],a_{i+1}=f[v_{i+1}],\cdots,a_k=f[v_k]))$$

$$g[(\text{tag}=t,\text{value}=(a_1=v_1,a_2=v_2,\cdots,a_{i-1}=v_{i-1},a_i=v,a_{i+1}=v_{i+1},\cdots,a_k=v_k))]$$
$$=(a_1=g[v_1],a_2=g[v_2],\cdots,a_{i-1}=g[v_{i-1}],a_i=(\text{tag}=t,\text{value}=g[v]),$$
$$a_{i+1}=g[v_{i+1}],\cdots,a_k=g[v_k])$$

Remarks

This transformation has the value of exposing unions to other transformations that eliminate unions. In particular, in combination with the preceding and following transformations, this transformation reduces the use of union constructors in most types.

4.3.4. Eliminate a Union that is inside a Set

When a set has elements from a union, the union is eliminated by transforming the set to a tuple with an attribute for each type of the union. The domain of each attribute is a set on the appropriate type. For example, replace the expression $\{(f:real|g:integer)\}$ by $(f:\{real\},g:\{integer\})$. This transformation is based on Hull and Yap's [HuY84] *collect-class* reduction.

Formal Definition

Given an expression $\{(t_1:d_1|t_2:d_2|\cdots|t_k:d_k)\}$,

replace the expression by $(t_1:\{d_1\},t_2:\{d_2\},\cdots,t_k:\{d_k\})$.

Proof of information preservation

Specify the generic functions used to construct the data transformation functions.

$$f[\{v_1,v_2,\cdots,v_p\}]=(t_1=\overline{f}[\{v_1,v_2,\cdots,v_p\},t_1],t_2=\overline{f}[\{v_1,v_2,\cdots,v_p\},t_2],$$
$$\cdots,t_k=\overline{f}[\{v_1,v_2,\cdots,v_p\},t_k])$$

$$\overline{f}[\{v_1,v_2,\cdots,v_p\},t]=\{f[v]\mid(\text{tag}=t,\text{value}=v)\in\{v_1,v_2,\cdots,v_p\}\}$$

$$g[(t_1=set_1,t_2=set_2,\cdots,t_k=set_k)]=\bigcup_{i=1}^{n}\{(\text{tag}=t_i,\text{value}=g[v])\mid v\in set_i\}$$

4.3.5. Simplify a Single-attribute Tuple

A tuple having a single attribute is transformed to the attribute's domain. For example, replace the expression (*f*:*real*) by *real*. This transformation is based on Hull and Yap's *comp-x* reduction [HuY84].

Formal Definition

Given an expression $(a_1{:}d_1)$, replace the expression by d_1.

Proof of information preservation

Specify the generic functions used to construct the data transformation functions.

$$f[(a_1 = v)] = f[v]$$

The form of g is determined by the type of the attribute's domain. It is the appropriate one from the following group.

$$g[(a_{1_1} = v_1, a_{1_2} = v_2, \cdots, a_{1_k} = v_k)] = (a_1 = (a_{1_1} = g[v_1], a_{1_2} = g[v_2], \cdots, a_{1_k} = g[v_k]))$$

$$g[(\text{tag} = t, \text{value} = v)] = (a_1 = (\text{tag} = t, \text{value} = g[v]))$$

$$g[\{v_1, v_2, \cdots, v_k\}] = (a_1 = \{g[v_1], g[v_2], \cdots, g[v_k]\})$$

$$g[<v_1, v_2, \cdots, v_k>] = (a_1 = <g[v_1], g[v_2], \cdots, g[v_k]>)$$

$$g[x] = (a_1 = x) \quad (x \text{ is an element of a basic type})$$

4.3.6. Remove Simple Recursion from a Type

If a type is simply recursive and in an appropriate form, it is transformed to a tuple that has less simple recursion. The resulting tuple has two attributes, one is a sequence that holds the recursive part and the other is a value for the rest of the information. A type is in an appropriate form if it is a two-domain union with one domain that is a two-attribute tuple with one attribute's domain being the type. For example, the type definition $n=(t1:(a1:real, a2:n)|t2:string)$ is in an appropriate form. The transformation replaces that type definition by the equivalent one $n=(sequence=<real>, final=string)$.

Formal Definition

Given a type definition $n=(t_1:(a_1:d_1, a_2:n)|t_2:d_2)$,

replace the type definition by $n=(sequence:<d_1>, final:d_2)$.

Proof of information preservation

Specify the generic functions used to construct the data transformation functions.

$f[v]=(sequence=f_s[v], final=f_v[v])$

$f_v[(\text{tag}=t, \text{value}=v)]=$ if $t=t_1$ then $f_v[v]$ else $f[v]$

$f_s[(\text{tag}=t, \text{value}=v)]=$ if $t=t_1$ then $\overline{f}_s[v]$ else $<>$

$\overline{f}_s[(a_1=v_1, a_2=v_2)]=insert[f[v_1], f_s[v_2]]$

$g[(sequence=<>, final=v_f)]=(\text{tag}=t_2, \text{value}=g[v_f])$

$g[<v_1, v_2, v_3, \cdots, v_p>, v_f]$

$=(\text{tag}=t_1, \text{value}=(a_1=g[v_1], a_2=g[(sequence=<v_2, v_3, \cdots, v_p>, final=v_f)]))$.

Remarks

The strict requirements of the appropriate form of a type may seem to limit application of this reduction. This is not the case since the other transformations and their inverses can transform any simply recursive type to one in an appropriate form. A more general definition of appropriate form could be given. With that definition, any simply recursive type can be transformed to one in an appropriate form by three of the basic transformations given above. The given definition is preferred because it is consistent with the intention of presenting only basic transformations and it makes the rest of this section comprehensible.

4.3.7. Combine Identical Domains of a Union

A union with two domains of the same type is transformed to a union with these two domains combined into one domain. This new domain is a tuple with a boolean attribute to indicate the original tag and another attribute for the value. For example, replace the expression $(a{:}real|b{:}string|c{:}string)$ by the expression $(a{:}real|b{:}(switch{:}boolean,value{:}string))$. There is a complication in transforming a union of two domains. If one is eliminated, the result can not be a union of one domain since that is invalid. Therefore as a special case, replace a union that has two identical domain by a tuple constructed in the manner described above.

Formal Definition

Given an expression $(t_1{:}d_1|t_2{:}d_2|\cdots|t_k{:}d_k)$

where for some i and j, $1 \le i < j \le k$, d_i and d_j are equivalent expressions

and assuming without loss of generality that $i = 1$ and $j = 2$,

replace the expression by $(switch:boolean, value:d_1)$ when $k=2$ and otherwise by

$$(t_1:(switch:boolean, value:d_1)|t_3:d_3|t_4:d_4|\cdots|t_k:d_k).$$

Proof of information preservation

Use *true* to indicate that the original tag was t_i and *false* for t_j.

Remarks

If intervals of integers were basic types of the semilattice data model, this transformation could be generalized to combine many identical domains. In that case, it may no longer be possible to find data transformation functions that manipulate data in an appropriate manner. The last section of this chapter discusses a transformation with that problem.

4.3.8. Combine Similar Domains of a Union

A union with two similar domains is transformed to a union with one domain replacing these two domains. The first domain may be any type. The second domain must be a three-attribute tuple whose attributes' domains are the first domain, some third domain, and a sequence on the third domain. The new domain is a two-attribute tuple whose domains are the first domain and a sequence on the third domain. An empty sequence indicates a value from the first domain. For example, replace the expression $(t1:real|t2:(e:real,f:string)|t3:(a:(e:real,f:string),b:integer,c:<integer>))$ by $(t1:real|t3:(a:(e:real,f:string),c:<integer>))$. Again as a special case, replace a two-domain union by the new domain. For the sake of brevity, only the description for the general case is given.

Formal Definition

Given an expression $(t_1:d_1|t_2:d_2|\cdots|t_k:d_k)$

where for some i and j, $1 \le i,j \le k$, d_j is $(a_{j_1}:d_{i_1}, a_{j_2}:d_{j_2}, a_{j_3}:<d_{j_2}>)$

and assuming without loss of generality that $j = i+1$,

replace the expression by

$$(t_1:d_1|t_2:d_2|\cdots|t_{i-1}:d_{i-1}|t_j:(a_{j_1}:d_i, a_{j_3}:<d_{j_2}>)|t_{i+2}:d_{i+2}|\cdots|t_k:d_k).$$

Proof of information preservation

Specify the generic functions used to construct the data transformation functions. f uses the empty sequence for t_i values and inserts a_{j_2} onto the sequence for t_j values.

$$f[(\text{tag}=t,\text{value}=v)] = \text{if } t = t_i \text{ then } (\text{tag}=t_j,\text{value}=(a_{j_1}=f[v], a_{j_3}=<>))$$
$$\text{else if } t = t_j \text{ then } (\text{tag}=t_j,\text{value}=f_j[f[v]])$$
$$\text{else } (\text{tag}=t,\text{value}=f[v])$$

$$f_j[(a_{j_1}=v_1, a_{j_2}=v_2, a_{j_3}=v_3)] = (a_{j_1}=v_1, a_{j_3}=\mathit{insert}[v_2,v_3])$$

$$g[(\text{tag}=t,\text{value}=v)] = \text{if } t = t_j \text{ then } \overline{g}[g[v]] \text{ else } (\text{tag}=t,\text{value}=g[v])$$

$$\overline{g}[(a_{j_1}=v_1, a_{j_3}=<>)] = (\text{tag}=t_i,\text{value}=v_1)$$

$$\overline{g}[(a_{j_1}=v_1, a_{j_3}=<v_2,v_3,\cdots,v_p>)]$$
$$= (\text{tag}=t_j,\text{value}=(a_{j_1}=v_1, a_{j_2}=v_2, a_{j_3}=<v_3,v_4,\cdots,v_p>))$$

4.3.9. Split an Unreferenced Union Type

If a type definition is not referenced by any type definition (including itself) and its primary constructor is a union, the type is replaced by a group of types that are more simply defined. This group has a type for each domain of the union. This transformation is information-preserving since there is an implicit union of all types in the database schema. For example, a database schema

$$\{s=(a:real|b=(e:real,f:string)|c=(e:real,f:string,g:integer,h:<integer>))\}$$

is transformed to

$$\{s_a=real, s_b=(e:real,f:string), s_c=(e:real,f:string,g:integer,h:<integer>)\}.$$

Formal Definition

Given a type definition $n=(t_1:d_1|t_2:d_2|\cdots|t_k:d_k)$

where n is not referenced by any type in the schema,

replace the type definition by $n_1=d_1,\ n_2=d_2,\ \cdots,\ n_k=d_k$

where n_j is the concatenation of n, "_", t_j and enough "_"'s to make it a unique type name in the schema.

Proof of information preservation

Specify the generic functions used to construct the data transformation functions. f maps from one type to many. g maps from many types to one, and for each different type it assigns the appropriate tag.

$$f[(\text{tag}=t,\text{value}=v)]=v \qquad g[v]=(\text{tag}=t_i,\text{value}=v)$$

4.3.10. Combine Two Sequences in a Tuple

A tuple with two sequences is transformed to a tuple by one sequence when a sequence of the first type is a part of every element of the other sequence. The new sequence is on a union between the elements of the first sequence and the elements of the second sequence without the part that has the first sequence as a domain. For example, replace the expression $(b:<integer>,c:<(e:string,f:<integer>)>)$ by $(c:<(b:integer\,|\,c:string)>)$. For clarity, the transformation described in Section 4.3.6 requires that a type is in an appropriate form before the transformation can be applied. For this transformation, the formal description requires that the expression is in an appropriate form. An expression is in an appropriate form if the enclosing sequence is a two-attribute tuple.

Formal Definition

Given an expression $(a_1:d_1, a_2:d_2, \cdots, a_k:d_k)$

where for some i and j, $1 \le i,j \le k$, d_i is $<d_{i_1}>$ and d_j is $<(a_{j_1}:d_{j_1}, a_{j_2}:<d_{i_1}>)>$

and assuming without loss of generality that $i=1$ and $j=2$,

replace the expression by $(a_2:<(a_1:d_{1_1}\,|\,a_2:d_{2_1})>, a_3:d_3, a_4:d_4, \cdots, a_k:d_k)$.

Proof of information preservation

Specify the generic functions used to construct the data transformation functions.

$$f[(a_1=v_1, a_2=v_2, a_3=v_3, \cdots, a_k=v_k)] = (a_2=\overline{f}[v_1,v_2], a_3=f[v_3], a_4=f[v_4], \cdots, a_k=f[v_k])$$

$$\overline{f}[<>,<>] = <>$$

$$\overline{f}[<v_1,v_2,\cdots,v_l>,v_{l+1}] = insert[(\text{tag}=a_1, \text{value}=f[v_1]), \overline{f}[<v_2,v_3,\cdots,v_l>,v_{l+1}]]$$

$$\overline{f}[<>,<(a_{2_1}=v_{2_1}, a_{2_2}=v_{2_2}), v_3, v_4, \cdots, v_l>] = insert[(\text{tag}=a_2, \text{value}=f[v_{2_1}]), \overline{f}[v_{2_2}, <v_3,v_4,\cdots,v_l>]]$$

$g[(a_2 = v_2, a_3 = v_3, a_4 = v_4, \cdots, a_k = v_k)]$

$= (a_1 = \overline{g}_1[g[v_2]], a_2 = \overline{g}_2[g[v_2]], a_3 = g[v_3], a_4 = g[v_4], \cdots, a_k = g[v_k])$

$\overline{g}_1$ builds a sequence of the leading a_1's

$\overline{g}_1[<(\text{tag} = a_1, \text{value} = v_1), v_2, v_3, \cdots, v_l>] = insert[v_1, \overline{g}_1 <v_2, v_3, \cdots, v_l>]$

$\overline{g}_1[<(\text{tag} = a_2, \text{value} = v_1), v_2, v_3, \cdots, v_l>] = <>$

$\overline{g}_1[<>] = <>$

$\overline{g}_2$ rebuilds the original a_2 sequence. For each a_2 in the sequence, $\overline{g}_2$ uses $\overline{g}_1$ to rebuild the sequence value for the second attribute of the tuple.

$\overline{g}_2[<(\text{tag} = a_1, \text{value} = v_1), v_2, v_3, \cdots, v_l>] = \overline{g}_2[<v_2, v_3, \cdots, v_l>]$

$\overline{g}_2[<(\text{tag} = a_2, \text{value} = v_1), v_2, v_3, \cdots, v_l>]$

$= insert[(a_{2_1} = v_1, a_{2_2} = \overline{g}_1[<v_2, v_3, \cdots, v_l>]), \overline{g}_2[<v_2, v_3, \cdots, v_l>]]$

$\overline{g}_2[<>] = <>$

4.4. Corrective Information-preserving Transformations

The descriptions of the corrective information-preserving transformations are in this section. In some sense, these transformations correct badly designed databases. They do not apply to reasonably designed databases but do apply to some syntactically correct schemas. Recall the earlier result that it can be determined if a certain type is valueless. This is it has the empty domain with respect to a particular schema. This knowledge is used to eliminate all of the valueless types of a schema and simplify any other types of the schema that refer to a valueless type. As a group, these transformations remove all parts of a schema that can not generate values.

4.4.1. Eliminate an Unreferenced Valueless Type

Any valueless type is eliminated if no other type references it.

Formal Definition

Drop a type $n = d$ such that $dom[S,n] = \emptyset$ and no other type definition references it.

Proof of information preservation

Trivial

4.4.2. Simplify a Tuple with the Empty Domain

If a tuple has an attribute whose domain is a valueless type, the tuple has the empty domain. Therefore transforming the tuple to the domain of the attribute is information preserving.

Formal Definition

Given an expression $(a_1:d_1, a_2:d_2, \cdots, a_k:d_k)$

where for some i, $1 \leq i \leq k$, d_i is n, the name of a type, and $dom[S,n] = \emptyset$,

replace the expression by n.

Proof of information preservation

Trivial

4.4.3. Simplify a Set on a Valueless Type

A set whose elements are from a valueless type has only one value (the empty set) and therefore is transformed to the singleton domain.

Formal Definition

Given an expression $\{n\}$

where n is the name of a type and $dom[S,n]=\emptyset$,

replace the expression by *singleton*.

Proof of information preservation

Trivial

4.4.4. Simplify a Sequence on a Valueless Type

A sequence whose elements are from a valueless type has only one value (the empty sequence) and therefore is transformed to the singleton domain.

Formal Definition

Given an expression $<n>$

where n is the name of a type and $dom[S,n]=\emptyset$,

replace the expression by *singleton*.

Proof of information preservation

Trivial

4.4.5. Drop an Domain on a Valueless Type from a Union

Similarly, a union with a domain that is on a valueless type is transformed to union without that domain. This transformation is information preserving since the union is never an element from that domain. In the special case that the union has only one other domain, replace the union by that other domain.

Formal Definition

Given an expression $(t_1:d_1|t_2:d_2|\cdots|t_k:d_k)$

where for some i, $1\leq i\leq k$, d_i is n, the name of a type, and $dom[S,n]=\emptyset$,

replace the expression by d_{3-i} when $k=2$ and otherwise by

$$(t_1{:}d_1|t_2{:}d_2|\cdots|t_{i-1}{:}d_{i-1}|t_{i+1}{:}d_{i+1}|\cdots|t_k{:}d_k).$$

Proof of information preservation

Trivial.

4.4.6. Remove a Singleton Attribute from a Tuple

If a tuple has an attribute with the singleton domain and other attributes, then replace the tuple by one with only the other attributes.

Formal Definition

Given an expression $(a_1{:}d_1, a_2{:}d_2, \cdots, a_k{:}d_k)$

where $k \geq 2$ and for some $1 \leq i \leq k$, d_i is *singleton*,

replace the expression by $(a_1{:}d_1, a_2{:}d_2, \cdots, a_{i-1}{:}d_{i-1}, a_{i+1}{:}d_{i+1}, \cdots, a_k{:}d_k)$.

Proof of information preservation

Specify the generic functions used to construct the data transformation functions.

$$f[(a_1 = v_1, a_2 = v_2, \cdots, a_{i-1} = v_{i-1}, a_i = \gamma, a_{i+1} = v_{i+1}, \cdots, a_k = v_k)]$$
$$= (a_1 = f[v_1], a_2 = f[v_2], \cdots, a_{i-1} = f[v_{i-1}], a_{i+1} = f[v_{i+1}], \cdots, a_k = f[v_k])$$
$$g[(a_1 = v_1, a_2 = v_2, \cdots, a_{i-1} = v_{i-1}, a_{i+1} = v_{i+1}, \cdots, a_k = v_k)]$$
$$= (a_1 = g[v_1], a_2 = g[v_2], \cdots, a_{i-1} = g[v_{i-1}], a_i = \gamma, a_{i+1} = g[v_{i+1}], \cdots, a_k = g[v_k])$$

4.4.7. Change a Set on the Singleton Domain to a Boolean

Any set with all elements from the singleton domain is transformed to a boolean domain.

Formal Definition

Given an expression {*singleton*}, replace the expression by *boolean*.

Proof of information preservation

Use *true* to indicate the presence of the singleton element and *false* for the empty set.

4.5. Final Remarks

A large group of information-preserving transformations is generated from the transformations presented in this chapter. It is felt that each transformation in the group treats data in an appropriate manner. It is also felt that the group contains all known transformations that treat data in an appropriate manner.

Traditional normalization, as described by Codd in [Cod70], is not a composition of the transformations described in this chapter. The transformation of Section 4.3.1, *include a subordinate tuple's attributes*, is part of traditional normalization. The remainder of normalization is not information preserving. A type representing a non-first normal form relation may need to be constrained in order that normalization preserve the data. For instance, normalization of the type $n=\{(a{:}real,b{:}\{string\})\}$ gives the type $n=\{(a{:}real,b{:}string)\}$. The corresponding data transformation function is one-to-one if and only if the non-first normal form relation has a functional dependency $a \rightarrow b$ and the empty set is not permitted as a value of b.

Another information-preserving schema transformation was found and was included in the preliminary drafts of this chapter. This transformation replaces any expression $<singleton>$ by the expression *natural*. One data transformation function determines the length of a sequence. The other produces a sequence $<\gamma,\gamma,\cdots,\gamma>$ of length n for any value of n. These functions appear to treat data in an appropriate manner but it was eventually decided that the transformation should not be included. This decision was made because the implications of including it were unacceptable. To illustrate the problems, the expression $(t1{:}(a{:}<singleton>,b{:}singleton,c{:}real)|t2{:}real)$ is considered. The transformations *combine similar domains or a union* and *remove a singleton attribute from a tuple* apply to this expression. If the two domains are

combined and then the expression *<singleton>* is replaced, the result is (*a*:*natural*,*c*:*real*). On the other hand, removing the singleton attribute and then replacing the expression *<singleton>* gives (*t*1:(*a*:*natural*,*c*:*real*)|*t*2:*real*). Recall that the inverse of an information-preserving transformation is information preserving and that the composition of information-preserving transformations is information preserving. Therefore, it is information preserving to combine the two domains in (*t*1:(*a*:*natural*,*c*:*real*)|*t*2:*real*). A value of zero for the attribute *a* would indicate a value from the second domain. A value of *i+1* indicates a value from the first domain by *a*'s actual value being *i*. It is intended that this chapter's transformations only generate transformations that treat data in an appropriate manner. By the criteria given at start of the chapter, a data transformation function

"should treat data values as essentially uninterpreted objects."

For this criteria, it is difficult to accept the function given above. If it is accepted then many other coding functions must also be accepted.

Chapter 5

Normalizing Semilattice Schemas

This chapter documents the evolution of a finite Church-Rosser replacement system that normalizes database schemas of the semilattice data model. From the outset of the research for this thesis, the goal was to incorporate all information-preserving transformations that had appropriate data transformation functions into a finite Church-Rosser replacement system. A finite Church-Rosser replacement system was desired because in other situations, replacement systems with this property are of great value. The classic case is the understanding of functions gained by describing them by expressions in λ-calculus and reducing them using Church and Rosser's reduction system.

The chapter begins with a section on replacement systems. It gives precise definitions of some terms, including replacement system, finite, and Church-Rosser. The second section of the chapter presents the desired replacement system, one including the information-preserving transformations of Sections 4.2, 4.3, and 4.4. Examples are given showing that this replacement system is neither finite nor Church-Rosser. Next is a discussion of some attempts to modify the replacement system to make it finite Church-Rosser.

In the third section of the chapter, another replacement system is defined. This is followed by a description of a new general method of proving that a replacement system is finite. A proof based on this method is given that the replacement system is finite. Finally, it is shown that the replacement system is finite Church-Rosser. The proof is based on Sethi's method [Set74].

The concluding section of the chapter looks at the difficulty of determining if two schemas are equivalent and presents some observations about normal form schemas of the Church-Rosser replacement system.

5.1. Replacement Systems

These definitions are from Sethi's work on finite Church-Rosser replacement systems [Set74]. An infix notation is used for all binary relations, thus $x R y$ means that $(x,y) \in R$.

A *replacement system* $(S, \Rightarrow, =)$ consists of a set S, the *replacement* relation $\Rightarrow$, any binary relation on S, and any equivalence relation $=$ on S. Typically, if $x \Rightarrow y$ then y is x with some simplification and thus it is said that x reduces to y. Ideally, the identity relation should be used as the equivalence relation but when this is impossible a relatively sparse relation is acceptable. For instance, in lambda calculus, the names of variables are insignificant in the evaluation of an expression and therefore, the expressions $\lambda x.x$ and $\lambda y.y$ are equivalent. Many replacement systems change elements that are expressions by replacing a term by an equivalent but simpler term. For this reason, some authors refer to *term rewriting systems* rather than replacement systems.

An element x of the set of a replacement system is *irreducible* or in *normal form* if there is no y such that $x \Rightarrow y$. Various relations are generated from the replacement relation and the following notation is used.

$x \Rightarrow^i y$ when $x \Rightarrow x_1$ and $x_1 \Rightarrow x_2$ and $\cdots$ and $x_{i-1} \Rightarrow y$

$x \Rightarrow^0 y$ when $x = y$

This relation is the identity relation.

$x \Rightarrow^* y$ when $x \Rightarrow^i y$ for some $i \geq 0$

This relation is the reflexive and transitive closure of $\Rightarrow$.

$x \Rightarrow \bar{} y$ when $x \Rightarrow^* y$ and y is an irreducible element

This relation is called the *completion* of $\Rightarrow$.

A replacement system is *Church-Rosser* if whenever $w = x$, $w \Rightarrow \bar{} y$ and $x \Rightarrow \bar{} z$ then $y = z$. A replacement system is *finite* or it *terminates* if for any element x there is

a bound k such that the set $\{y | x \Rightarrow^k y\}$ is empty. In particular, for no x is there a cycle $x \Rightarrow x_1 \Rightarrow x_2 \Rightarrow \cdots \Rightarrow x_i \Rightarrow x$. Also, there is no series $x \Rightarrow x_1 \Rightarrow x_2 \Rightarrow \cdots$ with all elements distinct. A replacement system is *finite Church-Rosser* or *confluent* if it is both finite and Church-Rosser.

5.2. The Desired Replacement System

This section describes a replacement system. Before that can be done, it is necessary to define another equivalence relation on semilattice database schemas. Recall that Section 4.1 defined informational equivalence (denoted by $\equiv_I$.) That definition allows arbitrary manipulations of the data and treats as equivalent many schemas that have no intuitive relationship. Therefore, a more useful definition of equivalence based only on the transformations described in Section 4.2, 4.3 and 4.4 is used in the remainder of this thesis. It is felt that these transformations change the data in an appropriate manner only. The relation denoted $\equiv_A$ is the equivalence relation generated from the union of the transformations of Section 4.2, 4.3 and 4.4. If $S \equiv_A S'$, it is said that S and S' are *informationally equivalent in the allowed manner.*

A replacement system with certain properties is desired. The system must use all of the transformations of Chapter 4. The system should be finite Church-Rosser. It should produce the same normal form for S and S' if and only if $S \equiv_A S'$. It is not possible to achieve all of these goals in one system. This section describes a replacement system that used all of the transformation of Chapter 4. Although this replacement system has many desirable features, it is unfortunately neither finite nor Church-Rosser.

The transformation of Section 4.2.3 *change a type's name* can always be applied to any schema. The transformation of Section 4.2.4 *change a tag or attribute name* can always be applied to any schema with a tuple or union constructor. If these transformation are reductions, some arbitrary method of choosing names must be incorporated

for any schema to have a corresponding normal form. Therefore the union of these two transformations generate the equivalence relation of the replacement system. The equivalence relation of this replacement system is called *structural equivalence* and denoted by $=_S$. If $S =_S S'$, it is said that S and S' are *structurally equivalent*.

The replacement relation of the system is the union of all of the other transformations of Chapter 4. An example of a valid series of reductions for the database schema $\{n1=(t11:n2|t12:real),n2=(a21:n1,a22:string)\}$ is given below. The section number of the transformation that produced the reduction is noted next to the right-hand margin.

$\{n1=(t11:(a21:n1,a22:string)|t12:real),n2=(a21:n1,a22:string)\}$ 4.2.1

$\{n1=(sequence:<string>,final:real),n2=(a21:n1,a22:string)\}$ 4.3.6

$\{n1=(sequence:<string>,final:real),$ 4.2.1
$n2=(a21:(sequence:<string>,final:real),a22:string)\}$

$\{n1=(sequence:<string>,final:real),$ 4.3.1
$n2=(sub_sequence:<string>,sub_final:real,sup_a22:string)\}$

5.2.1. How the Desired Replacement System Fails to be Finite

Substituting a type's definition for its name has positive effects in many situations, but it permits infinite series of reductions in some schemas. For instance, substitution has no effect on a type definition like $n=n$ and thus could applied arbitrarily often. The database schema $\{n1=(a11:integer,a12:n2),\ n2=(a21:real,a22:\{n2\})\}$ is a less contrived example but nonetheless, a substitution for $n2$ in $n1$ can be repeated over and over again. Arbitrarily involved examples can be developed since an infinite series of reductions exists for any schema that has recursive types.

Therefore, in order to have a finite replacement system, different replacement relations that restrict substitution were considered. Never permitting substitution of a recursive schema was the obvious solution. Unfortunately that restriction makes the

initial schema of the example of the previous section irreducible and yet, as was demonstrated, a reasonable series of reductions is possible. Therefore a different restriction on substitution was developed. Substitution of a type's definition into a second type is permitted when the first type is not recursive in a manner independent of the second type. It can be shown that the replacement system is finite if substitution is restricted in this manner.

5.2.2. How the Desired Replacement System Fails to be Church-Rosser

The desired replacement is not Church-Rosser because it produces more than one irreducible form for many database schemas. A typical problem can be demonstrated with the schema $\{n=\{(t1:real|t2:real)\}\}$. Applying the transformation of Section 4.3.4, *eliminate a union that is inside a set*, yields $\{n=(t1:\{real\},t2:\{real\})\}$. On the other hand, the union in the original schema could also be transformed by the transformation of Section 4.3.8, *combine identical domains in a union*. Since the union has only two domains, combining the two identical domains of the union gives $\{n=\{(switch:boolean,value:real)\}\}$. For either schema, no further reductions are possible and therefore the system is Church-Rosser only if the two schemas are structurally equivalent. Since they are not, the replacement system is not Church-Rosser.

A way was found to get the desired behavior in this and similar situations. A reduction was defined for tuples that combines two attributes of a tuple when both are set domains with the same element domain. This reduction is appropriate because it is the composition of existing reductions and their inverses.

For other situations, it was also necessary to define reductions that are series of transformations. Typically these require a few inverse transformations and then a single transformation. Two transformations have many problems because their application is highly restricted. The transformation *remove simple recursion from a type* was generalized and its definition is elaborated later. The other problem transformation,

combine two sequences in a tuple, was also generalized. In the general form, it can apply in any situation where one sequence is *part of* the elements of the other. A formal definition of *part of* is omitted since it is involved and as is shown later, with any definition, the transformation is unsuitable for a Church-Rosser replacement system. In the rest of this thesis, the more general definitions of transformations are used. The rest of this section contains examples of problems that are not correctable by defining new reductions.

5.2.2.1. Substitute a Type's Definition for Its Name

In the database schema $\{n1=(a{:}string,b{:}\{n2\}),n2=(a{:}real,b{:}\{n1\})\}$, either type definition may be substituted into the other. Substituting the definition of $n2$ into $n1$ gives $\{n1=(a{:}string,b{:}\{(a{:}real,b{:}\{n1\})\}),n2=(a{:}real,b{:}\{n1\})\}$. The database schema $\{n1=(a{:}string,b{:}\{n2\}),n2=(a{:}real,b{:}\{(a{:}string,b{:}\{n2\})\})\}$ results from substituting the other way. No further reductions are possible for either schema and the two are not structurally equivalent. Therefore, the replacement system is not Church-Rosser.

Interestingly, when both substitutions are done at the same time, the result is $\{n1=(a{:}string,b{:}\{(a{:}real,b{:}\{n1\})\}),n2=(a{:}real,b{:}\{(a{:}string,b{:}\{n2\})\})\}$. This is an equivalent schema where neither type now refers to the other. Clearly, only the original schema or the one from mutual substitution is acceptable as the normal form in a Church-Rosser replacement system.

5.2.2.2. Combine Similar Domains in a Union

The transformation *combine similar domains in a union* was discovered to make the replacement system have the appropriate behavior for a class of schemas. This section gives an example of a schema where the transformation is required to make all series of reductions give isomorphic results. After that, three examples are given of problems it causes.

The transformation is required for the database schema

$\{n1=(t11:(a11:string,a12:n2)|t12:integer),n2=(t21:(a21:real,a22:n1)|t22:boolean)\}$.

For this schema, the only two reductions applicable are substitutions of either type definition into the other. A complete series of reductions is given below. The section number of the transformation is listed beside the right margin. This is an involved example and therefore in interest of clarity, two deviations from rigorous adherence to the given definitions of the transformations are made. Only one reduction does tag renaming although several other actually should. Also, since each reduction only changes one type definition, only the changed definition is shown.

Starting with the substitution of the type definition of $n2$ into $n1$, all the reductions to $n1$'s definition are given first.

$n1=(t11:(a11:string,a12:(t21:(a21:real,a22:n1)|t22:boolean))$
$|t12:integer)$ 4.2.1

$n1=(t11:(t21:(a11:string,a12:(a21:real,a22:n1))|t22:(a11:string,a12:boolean))$
$|t12:integer)$ 4.3.3

$n1=(t21:(a11:string,a12:(a21:real,a22:n1))|t22:(a11:string,a12:boolean)$
$|t12:integer)$ 4.3.2

$n1=(t21:(a11:string,a21:real,a22:n1)|t22:(a11:string,a12:boolean)$
$|t12:integer)$ 4.3.1

$n1=(sequence:<(a11:string,a21:real)>,choice:(t22:(a11:string,a12:boolean)$
$|t12:integer))$ 4.3.6

$n1=(t22:(sequence:<(a11:string,a21:real)>,choice:(a11:string,a12:boolean))$
$|t12:(sequence:<(a11:string,a21:real)>,choice:integer))$ 4.3.3

$n1=(t22:(sequence:<(a11:string,a21:real)>,a11:string,a12:boolean)$
$|t12:(sequence:<(a11:string,a21:real)>,choice:integer))$ 4.3.1

This definition of $n1$ is irreducible. It is substituted into $n2$.

$n2$=($t21$:($a21$:*real*, 4.2.1
$a22$:($t22$:(*sequence*:<($a11$:*string*,$a21$:*real*)>,$a11$:*string*,$a12$:*boolean*)
|$t12$:(*sequence*:<($a11$:*string*,$a21$:*real*)>,*choice*:*integer*)))
|$t22$:*boolean*)

Now $n2$ is reduced.

$n2$=($t21$:($t22$:($a21$:*real*, 4.3.3
$a22$:(*sequence*:<($a11$:*string*,$a21$:*real*)>,$a11$:*string*,$a12$:*boolean*))
|$t12$:($a21$:*real*,$a22$:(*sequence*:<($a11$:*string*,$a21$:*real*)>,*choice*:*integer*)))
|$t22$:*boolean*)

$n2$=(*sub_t22*:($a21$:*real*, 4.3.2
$a22$:(*sequence*:<($a11$:*string*,$a21$:*real*)>,$a11$:*string*,$a12$:*boolean*))
|*sub_t12*:($a21$:*real*,$a22$:(*sequence*:<($a11$:*string*,$a21$:*real*)>,*choice*:*integer*))
|*sup_t22*:*boolean*)

$n2$=(*sub_t22*:($a21$:*real*, 4.3.1
sequence:<($a11$:*string*,$a21$:*real*)>,$a11$:*string*,$a12$:*boolean*)
|*sub_t12*:($a21$:*real*,$a22$:(*sequence*:<($a11$:*string*,$a21$:*real*)>,*choice*:*integer*))
|*sup_t22*:*boolean*)

Combining the similar domains gives the final irreducible form.

$n2$=(*sub_t22*:(*sequence*:<($a11$:*string*,$a21$:*real*)>,$a12$:*boolean*) 4.3.8
|*sub_t12*:($a21$:*real*,$a22$:(*sequence*:<($a11$:*string*,$a21$:*real*)>,*choice*:*integer*)))

A series of reductions starting with the other substitution is similar. If all alternative series of reductions are examined, the results are all structurally equivalent if and only if a *combine similar domains in a union* transformation is included in each series.

Unfortunately, this transformation prevents the replacement system from being Church-Rosser. Three examples that follow demonstrate this. Each example gives a

database schema with a single type definition. Two or three different reductions apply to this schema. Each of the reductions yields an irreducible schema. The system is not Church-Rosser since the results are not structurally equivalent. Since the first example only involves only *combine similar domains of a union* transformations, it suggests that the transformation does not belong to any Church-Rosser replacement system.

Example 1

Consider the database schema

$$\{n=(t1:(a:string,b:integer,c:<integer>)|t2:(a:string,b:real,c:<real>)|t3:string)\}.$$

The domain of $t3$ is similar to both of the other domains. Therefore, it may be combined with either of them. Combining it with the domains of $t1$ yields

$$\{n=(t1:(a:string,c:<integer>)|t2:(a:string,b:real,c:<real>))\}.$$

Combining the other way gives

$$\{n=(t1:(a:string,b:integer,c:<integer>)|t2:(a:string,c:<real>))\}.$$

No further reductions are possible in either case and the two forms are not structurally equivalent. Therefore, the replacement system is not Church-Rosser.

Example 2

Consider the database schema

$$\{n=(t1:(a:string,b:integer,c:<integer>)|t2:string|t3:string)\}.$$

Combining the similar domains of $t1$ and $t2$ yields

$$\{n=(t1:(a:string,c:<integer>)|t3:string)\}.$$

Combining the similar domains of $t1$ and $t3$ gives an equivalent form but combining the identical domains $t2$ and $t3$ gives

$\{n=(t1:(a:string,b:integer,c:<integer>)|t2:(switch:boolean,value:string))\}$.

This schema is irreducible and not equivalent to the other two forms which are also irreducible.

Example 3

Consider the database schema $\{n=(t1:(a:string,b:n,c:<n>)|t2:string|t3:real)\}$. The schema $\{n=(t1:(a:string,c:<n>)|t3:real)\}$ is produced by combining the similar domains of $t1$ and $t2$. The original type definition of n is simply recursive by the b attribute of the tuple that is $t1$'s domain. Eliminating this simple recursion gives the schema $\{n=(sequence<(a:string,c:<n>)>,final:(t2:string|t3:real))\}$. This can be further reduced to

$\{n=(t2:(sequence<(a:string,c:<n>)>,final:string),$
$|t3:(sequence<(a:string,c:<n>)>,final:real\ ))\}$.

Neither of the two schemas produced by eliminating simple recursion is structurally equivalent to the one produced by combining similar domains.

5.2.2.3. Combine Two Sequences in a Tuple

The transformation *combines two sequences in a tuple* was also discovered to give the replacement system appropriate behavior for a class of schemas. If a type has two simple recursions then they can be removed in either order and applying this transformation gives an equivalent schema. For example, the database schema $\{n=(t1:(a:integer,b:n)|t2:(a:real,b:n)|t3:string)\}$ reduces to

either $\{n=(sup_sequence:<(sequence:<integer>,a:real)>,$
$sub_sequence:<integer>,sub_choice:string)\}$
or $\{n=(sup_sequence:<(sequence:<real>,a:integer)>,$
$sub_sequence:<real>,sub_choice:string)\}$

when no *combines two sequences in a tuple* transformations are done. Combining the

two sequences of the first of these schemas gives

$$n=(sup_sequence:(sup_sequence:integer\,|\,sub_sequence:real),choice:string).$$

Combining the two sequences of the second schema gives a structurally equivalent schema.

As was seen in Example 3 of the previous section, for some database schemas, the transformation *combine two sequences in a tuple* in combination with others prevents the replacement system from having the Church-Rosser property. More seriously, this transformation causes problems without involving any other transformations. Consider $S1$, the database schema

$$\{n=(a1:<(a:integer,b:integer,c:<(a:string,b:string,c:<(a:real,b:real)>)>)>,\\ a2:<(a:string,b:string,c:<(a:real,b:real)>)>,\\ a3:<(a:real,b:real)>)\ \}.$$

For this schema, two reductions are possible. Once either is done no further reductions are possible and the two forms are not structurally equivalent. Combining the sequences of the $a1$ and $a2$ attributes yields

$$\{n=(a1:<(a1:(a:integer,b:integer)\,|\,a2:(a:string,b:string,c:<(a:real,b:real)>))>,\\ a3:<(a:real,b:real)>)\ \}.$$

Combining the sequences of the $a2$ and $a3$ attributes yields

$$\{n=(a1:<(a:integer,b:integer,c:<(a:string,b:string,c:<(a:real,b:real)>)>)>,\\ a2:<(a2:(a:string,b:string)\,|\,a3:(a:real,b:real))>)\ \}.$$

To demonstrate the complexity of this problem, two other schemas are presented and reduced. The reductions show that all schemas are informationally equivalent in the allowed manner but none are structurally equivalent. Consider $S2$, the database schema

$$\{n=(a1:<(a:integer,b:integer)>,\\ a2:<(a:string,b:string,c:<(a:integer,b:integer)>,d:<(a:real,b:real)>)>,\\ a3:<(a:real,b:real)>)\ \}.$$

Combining the sequences of the attributes $a1$ and $a2$ yields the same schema as the first reduction of $S1$ and thus shows $S1 =_A S2$. Combining the sequences of the attributes $a2$ and $a3$ yield

$$\{n=(a1:<(a:integer,b:integer)>,$$
$$a2:<(a2:(a:string,b:string,c:<(a:integer,b:integer)>)$$
$$|a3:(a:real,b:real))>)\}.$$

Now consider $S3$, the database schema

$$\{n=(a1:<(a:integer,b:integer)>,$$
$$a2:<(a:<(a:integer,b:integer)>,b:string,c:string)>,$$
$$a3:<(a:real,b:real,c:<(a:<(a:integer,b:integer)>,b:string,c:string)>)>)\}.$$

Combining the sequences of the attributes $a1$ and $a2$ yields a schema that is not structurally equivalent to any of those given above. This schema is

$$\{n=(a1:<(a1:(a:integer,b:integer)|a2:(a:string,b:string))>,$$
$$a3:<(a:real,b:real,c:<(a:<(a:integer,b:integer)>,b:string,c:string)>)>)\}.$$

Combining the sequences of the attributes $a2$ and $a3$ yields the same schema as the second reduction of $S2$. This shows $S2 =_A S3$ and therefore $S1 =_A S3$.

5.2.3. Making a Church-Rosser Replacement System

Many attempts were required to modify the desired replacement system to a finite one with the Church-Rosser property. Adding new basic transformations to replacement relation was and is the preferred method of modification but at this time, no other basic transformation have been found. If more are found, a proof that the resulting system is Church-Rosser will be more difficult since all possible combinations of reductions for all possible schemas must exhibit the desired behavior. Also, since the two transformations discovered to correct problems caused many other problems, it seems likely that other new basic transformations would also do this.

The replacement system behaves appropriately when the original schema does not have certain features. It would be possible to define a procedure to identify those

schemas that the replacement system always reduces appropriate. Therefore, a Church-Rosser replacement system for some schemas and all transformations could be built. This approach was rejected because there is no way to defend a claim that every *natural* database or conceivable application is included.

Enlarging the equivalence relation to the relation $=_A$ corrects all problems since then finiteness of the replacement relation guarantees that the Church-Rosser property holds. This is an absurd solution but it offered another method to modify the replacement system to have the desired properties. Since the replacement system could use any equivalence relation, it was hoped that perhaps some slight enlargement of structural equivalence would correct the problems. One difficulty with enlarging the equivalence relation is that it is more difficult to show the replacement system has the Church-Rosser properties. This follows from the requirement that all equivalent schemas must reduce to equivalent schemas. Another undesirable aspect of enlarging the equivalence relation is that the thesis's goal of understanding structure is comprised. Despite these drawbacks, two enlargements of the equivalence relation were tried.

One enlargement was generated from $=_S$ and the transformation *combine similar domains in a union*. This definition of equivalence meant that the schema $\{n=\langle(t1{:}real|t2{:}string)\rangle\}$ was equivalent to all schemas that are structurally equivalent to $\{n1=(t0{:}singleton|t1{:}(a1{:}(t1{:}real|t2{:}string),a2{:}\langle(t1{:}real|t2{:}string)\rangle))\}$. The second schema is equivalent to all schemas structurally equivalent to the schema

$$\{n2=(t0{:}singleton|t1{:}(a1{:}(t1{:}real|t2{:}string),\ a2{:}(t0{:}singleton|t1{:}(a1{:}(t1{:}real|t2{:}string),a2{:}\langle(t1{:}real|t2{:}string)\rangle)))))\}.$$

This expansion may be repeated over and over getting ever more complex types. Furthermore, in a finite Church-Rosser replacement system, whenever two elements are equivalent and the first one is irreducible then it must be equivalent to any irreducible element to which the second one reduces. This is a direct consequence of the finite Church-Rosser definition. Therefore, a further enlargement of equivalence

relation was made. This had terrible consequences. The schema $\{n=<(t1:real|t2:string)>\}$ is irreducible. The equivalence class of this schema includes any schema structurally equivalent to any schema produced by k replacements of a sequence sub-expression and all irreducible schemas to which these reduced. The equivalence class of the irreducible database schema $\{n=(a:real,b:<<(t1:real|t2:string)>>)\}$ was investigated briefly. It has bewildering structural diversity. The equivalence class of another irreducible schema $\{n=(a:real,b:<<(t1:(a:n,b:real)|t2:string)>>)\}$ is almost beyond description since this schema is equivalent to a schema with a simply recursive type. That schema reduces to an irreducible schema that is equivalent to a third schema with a simply recursive type. The third schema reduces to an irreducible schema that is equivalent to a fourth schema with a simply recursive type and so on.

This enlarged equivalence relation corrected most but not all of the problems described above. Even if it does produce a replacement system that is finite Church-Rosser, a replacement system with this equivalence relation is not acceptable. The notion of equivalence is so broad that the relation $=_A$ might as well be used. Another enlargement of the equivalence relation was generated from the transformation *combine two sequences in a tuple*. Although this equivalence relation had less prolific enlargements, it still was judged unacceptable for the same reason.

Finally, a finite Church-Rosser replacement system was built. Placing a restriction on substitution had produced a finite replacement system. Certainly, some of the problems appeared to result from not applying the correct transformation at the correct time. It was decided that restrictions were acceptable provided they were in the spirit of a replacement system. They had to be local, simple and intuitive. Nonetheless, since the restrictions increase the number of irreducible forms, simplicity was compromised to increase the normalizing power of the replacement system.

5.3. A Finite Church-Rosser Replacement System

This section documents the best finite Church-Rosser replacement system found. Since best is a relative term, the criteria by which the replacement system is best must be given. It was decided that the best system would use the structural equivalence relation $=_S$ because it is consistent with the thesis's goal of studying schema structure and the attempts to enlarge it had terrible consequences. Preference between different definitions of the replacement relation was given to the one that gave the fewest normal forms (modulo $=_S$) to the schemas of an $=_A$ equivalence class. All else being equal, the more intuitively appealing replacement relation was chosen.

5.3.1. The Replacement Relation

Table 5.1 on the following page defines the replacement relation. The name of a reduction is based on the section in Chapter 4 that describes its basic transformation. The reductions are identical to the transformations except as noted in the restriction's column of the table or as described below. Three of the basic transformations are not in the table because no reasonable restrictions could be found for them. These are the transformations *combine similar domains in a union, combine identical domains in a union,* and *combine two sequences in a tuple.*

Since two reductions are based on the substitution transformation, they are named 2.1a and 2.1b respectively. Reduction 2.1a is substitution restricted to substituting a non-recursive type. Separating a type, reduction 2.1b, is a related non-empty group of substitutions that is used to make a recursive type self-defining in some cases. Separation avoids the problems found with simple substitution but is restricted to avoid causing other problems.

Table 5.1 The Replacement Relation

Name	Description	Restriction
2.1a	Substitute a type's definition for its name	it is not a recursive type
2.1b	Separate a type	the type is separable
2.2	Replace two isomorphic types by one types	neither type is recursive
3.1	Include a subordinate tuple's attributes	
3.2	Include a subordinate union's domains	
3.3	Change a tuple with a union to a union of tuples	
3.4	Eliminate a union that is inside a set	
3.5	Simplify a single-attribute tuple	
3.6	Eliminate all simple recursion from a type	the type's form is appropriate
3.9	Split an unreferenced union type	
4.1	Eliminate an unreferenced valueless type	
4.2	Simplify a tuple with the empty domain	
4.3	Simplify a set on a valueless type	
4.4	Simplify a sequence on a valueless type	
4.5	Drop a domain on a valueless type from a union	
4.6	Remove a singleton attribute from a tuple	
4.7	Change a set on the singleton domain to a boolean	

Reduction 2.1b

Separating a type, reduction 2.1b, is a related non-empty group of substitutions that applies to some recursive types. A type is *separate* is it does not reference another type. To be *separable*, a type must not be unseparable. A type n is *unseparable* if it is separate or simply recursive or if it references a type that is part of a cycle $n_1, n_2, \cdots, n_k, n_1$ in the reference graph where $n_j \neq n$ for all j, $1 \leq j \leq k$.

A type is separated by substituting the definitions of other types into it until it no longer references any other type. In order to have a Church-Rosser replacement system, when a type is separated, it is necessary to also separate any other separable type that it references. It is essential when separating a group of types that the preseparation version of each type's definition is the one substituted into the other types.

The types $n1$ and $n3$ are separable in the semilattice database schema $\{n1=(t1:n2|t2:n3), n2=(a1:real, a2:n3), n3=\{(a1:integer, a2:n1, a3:\{n1\})\}\}$. The type $n2$ is unseparable because of the cycle between the other two schemas. The separation of $n1$ changes its definition and that of $n3$. The reduced schema is

$\{n1=(t1:(a1:real, a2:\{(a1:integer, a2:n1, a3:\{n1\})\}), |t2:\{(a1:integer, a2:n1, a3:\{n1\})\}),$
$n2=(a1:real, a2:n3),$
$n3=\{(a1:integer, a2:(t1:(a1:real, a2:n3), |t2:n3), a3:\{(t1:(a1:real, a2:n3), |t2:n3)\})\}\}.$

Reduction 2.2

Replacing two isomorphic types with one type, reduction 2.2, is an extension of the basic transformation of the same name. The reduction applies to more schemas since it combines a pair of types with isomorphic expressions as well as a pair with equivalent expressions. Two type expressions are isomorphic if a sequence of name changes made to one expression gives an expression equivalent to the other expression.

Reduction 3.6

Reduction 3.6 eliminates all simple recursion from a type that is in an appropriate form. A type is in an appropriate form when it is a union with at least one domain in each of two categories and every domain in one of these two categories. A domain is *properly recursive*, the first category, if it is the name of the type or a tuple with the type as the domain of an attribute. A domain is *non-recursive*, the second category, if it defined by an expression whose simplification does not mention the type.

To illustrate the definition of appropriate form, the type

$$n=(t1:n|t2:(a1:real,a2:string)|t3:(a1:string,a2:n)$$
$$|t4:(a1:n,a2:(t41:real|t42:n2))|t5:<string>)$$

is used. It is in an appropriate form for the elimination of all simple recursion. The domains $t1$, $t3$, and $t4$ are in the properly recursive category and the domains $t2$ and $t5$ are in the non-recursive category.

This definition of appropriate form is as general as possible without requiring additional restrictions on other reductions. Although the definition is restrictive, it is considerably more general than the preconditions for the application of the base transformation.

The reduction that eliminates all simple recursion in a type is a sequence of information-preserving transformations of Section 4.3. First, a sequence of inverse transformations produce a type that satisfies the preconditions of the base transformation *remove simple recursion from a type*. All simple recursion in the type is then removed by applying the base transformation.

The sequence of inverse transformations is:

1. For each recursive domain that is the name of the type, make it a single-attribute tuple by doing the inverse of the transformation *simplify a single-attribute tuple*.
2. For each recursive domain that is a single-attribute tuple, make it a two-attribute tuple by doing the inverse of the transformation *remove a singleton attribute from a tuple*.
3. For each recursive domain that is a many-attribute tuple, make it a two-attribute tuple by doing the inverse of the transformation *include a subordinate tuple's attributes*. In the resulting tuple, one attribute's domain is the type and the other attribute's domain is the tuple with all the other attributes of the original tuple.

4. When necessary to make the principal union have only one recursive domain, do the inverse of the transformation *change a tuple with a union to a union of tuples.*

5. When necessary, make the principal union have only one non-recursive domain by doing the inverse of the transformation *include a subordinate union's domain.*

For the definition of the type n given above, the sequence of inverse transformations produces

$$n = (recursive:(a:n, b = (t1:singleton \mid t3:(a1:string) \mid t4:(a2:(t41:real \mid t42:n2))))$$
$$\mid non-recursive:(t2:(a1:real, a2:string) \mid t5:<string>)).$$

After elimination of all simple recursion, the type definition is

$$n = (sequence:<(t1:(a:singleton) \mid t3:(a1:string) \mid t4:(a2:(t41:real \mid t42:n2)))>,$$
$$choice:(t2:(a1:real, a2:string) \mid t5:<string>)).$$

The Reasonability of the Reduction System

There are simple algorithms that given a database schema determine the possible reductions of the schema. Most of the reductions are local and a simple examination of each type definition can find these reductions. Reduction 2.2, replace two isomorphic types with one, requires each pair of type definitions be considered in turn. Examining a pair is linear in the length of the expressions since this is basically a tree isomorphism problem. Some reductions require particular properties of the types involved. These properties can be determined from the reference graph of a schema. This graph may be built in time linear to the length of the schema. A depth-first search of the graph starting at the node associated with a type can determine if the type is recursive or separable. The depth-first search need only examine each edge once for each node and therefore, the complexity of determining the properties for all nodes is $O(n^3)$ in the worst case.

5.3.2. A New Method to Prove Finiteness of a Replacement System

Proving that a system is finite Church-Rosser is usually difficult. In fact in the general case, it has been shown to be an undecidable problem [HuL78]. Nachum Dershowitz presented several techniques of showing that a replacement system is finite and gives a summary of related work in [Der82].

This thesis presents and uses a new method of proving the finiteness of a replacement system. It generalizes the technique that Hull and Yap used to prove termination of the replacement system of the Format data model[HuY84]. They specified a function α that maps formats into positive integers and showed that α has the property that if $f \Rightarrow g$ then $\alpha[f] > \alpha[g]$. Since all monotonically decreasing sequences of positive integers are finite, there can be no infinite series of reductions.

There are two steps in this new technique of showing that a replacement system is finite. First, an appropriate sequence of sets, partial orderings and functions are specified. Then Theorem 5.1 (given below) is applied. This method is more general than Hull and Yap's since it allows a sequence of functions rather than a single function. Also, the method allows the domains of the functions to be any set that has a well-founded partial ordering.

A partial ordering $>$ of a set S is *well-founded* if there does not exist an infinite sequence of elements such that $s_1 > s_2 > \cdots$.

Note that the definition of a well-founded partial ordering does not require the set be total-ordered or well-ordered by the partial ordering. For instance, the proper subset relation is a well-founded partial ordering of the finite power set of any set.

Allowing a sequence of functions is the more important generalization since then each function may concentrate on a different aspect of complexity. The relative importance of the different complexity aspects is fixed by the order of the sequence of functions. Some functions may actually increase some aspects of complexity for the

is permitted when a more important aspect is significantly reduced. Significance is guaranteed by requiring a well-founded partial ordering on the domain of each function in the sequence.

Since each function deals with only one aspect of complexity, the technique has the usual advantages of modularization. The functions are easy to conceive and specify since typically, a complexity aspect is easily measured by looking at syntactical features. A small change to the replacement system requires a small change in the proof. The proof of finiteness is more easily comprehended since it has many simple pieces rather than one incomprehensible function.

Theorem 5.1

A replacement system $(S,\Rightarrow,=)$ is finite if and only if there exists a sequence of sets $X_1,X_2,\cdots,X_k$ where each X_i has a well-founded partial ordering $>_i$ and a function $f_i:S\rightarrow X_i$ such that $\forall s,s'\in S$ if $s\Rightarrow s'$ then

$$\exists i\in\{1,2,\cdots,k\}\ ((f_i[s]>_i f_i[s'])\text{ and }\forall j\in\{1,2,\cdots,i-1\}\ (f_j[s]=f_j[s'])).$$

Proof (only if)

Given that $(S,\Rightarrow,=)$ is finite, let $f_1:S\rightarrow S$ be the identity function on S, and let $\Rightarrow$ be $>_1$. Trivially, $\forall s,s'\in S$ if $s\Rightarrow s'$ then $f_1[s]>_1 f_1[s']$.

Proof (if)

Given a replacement system $(S,\Rightarrow,=)$ and $X_1,X_2,\cdots,X_k$ where each X_i has a well-founded partial ordering $>_i$ and a function $f_i:S\rightarrow X_i$ such that $\forall s,s'\in S$ if $s\Rightarrow s'$ then

$$\exists i\in\{1,2,\cdots,k\}\ ((f_i[s]>_i f_i[s'])\text{ and }\forall j\in\{1,2,\cdots,i-1\}\ (f_j[s]=f_j[s'])).$$

Assume that the replacement system has an infinite sequence of reductions

$$s_0\Rightarrow s_1\Rightarrow s_2\Rightarrow\cdots.$$

Let $x \geq_i y$ denotes that $x >_i y$ or $x = y$. Note that

$$f_1[s_0] \geq_1 f_1[s_1] \geq_1 f_1[s_2] \geq_1 \cdots .$$

Since $>_i$ is a well-founded partial ordering of X_i, there exists n_1, such that $n_1 \geq 0$ and

$$f_1[s_{n_1}] = f_1[s_{n_1+1}] = f_1[s_{n_1+2}] = \cdots .$$

Note that

$$f_2[s_{n_1}] \geq_2 f_2[s_{n_1+1}] \geq_2 f_2[s_{n_1+2}] \geq_2 \cdots .$$

Using the same argument again, there exist n_2 such that $n_2 \geq n_1$ and

$$f_2[s_{n_2}] = f_2[s_{n_2+1}] = f_2[s_{n_2+2}] = \cdots .$$

Similarly, there exists $n_3, n_4, \cdots, n_k$ such that $n_k \geq n_{k-1} \geq \cdots \geq n_4 \geq n_3 \geq n_2$ and

$$\forall i \in \{1,2,\cdots,k\} \ (f_i[s_{n_i}] = f_i[s_{n_i+1}] = f_i[s_{n_i+2}] = \cdots).$$

Thus for all n and i, $n \geq n_k$ and $i \in \{1,2,\cdots,k\}$ $f_i[s_n] = f_i[s_{n+1}]$. Recall that $s_n \Rightarrow s_{n+1}$ and for any reduction $s \Rightarrow s'$ there is some i such that $f_i[s] >_i f_i[s']$ and therefore for some i, $f_i[s_n] >_i f_i[s_{n+1}]$.

This contradiction shows the assumption that an infinite series of reductions exists is false. The replacement system is finite.

5.3.3. Proof of Finiteness

The finiteness of the replacement system is proven by the method described above. Theorem 5.1 is applied after some appropriate functions are specified. These functions measure the complexity of a database schema by looking at syntactic properties. They were carefully constructed to give every reduction the property that Theorem 5.1 requires. Three functions map the schemas to sets of type names and use the proper subset relation as the partial ordering. The rest of the functions map schemas to non-negative integers and use the relation *less than* as the partial ordering.

Theorem 5.2

The replacement system of this section is finite.

Proof

For any database schema S, each of the following functions is well defined. (Specifications of the functions f_3 and f_4 follow the proof.)

$f_1[S]$ is the names of all types in S that are separable.

$f_2[S]$ is the names of the types in S that are simply recursive.

$f_3[S]$ is an upper bound on the number of *change a tuple with a union to a union of tuples* reductions in the longest sequence of reductions that apply to S and has no separations of types or eliminations of simple recursion.

$f_4[S]$ is a measure of the complexity of references to non-recursive types in S.

$f_5[S]$ is the total number of domains of all union constructors in S.

$f_6[S]$ is the total number of attributes of all tuple constructors in S.

$f_7[S]$ is the number of {*boolean*} sub-expressions in S.

$f_8[S]$ is the names of the types in S.

For the given replacement system, Table 5.2 below shows the effect on the values of the functions of applying the various types of reductions. In the table, the symbol = in row *x.y* and column f_i indicates that $f_i[S]=f_i[S']$ for any reduction $S \Rightarrow S'$ of type *x.y*. A > indicates the function's value after the reduction is always less than before it. If the value of the reduced schema is either the same or less, the symbol ≥ appears. For each type of reduction, no entries appear to the right of a > since the effect of the reduction on these functions is of no significance to the proof. Since the table shows that these functions have the necessary properties to apply Theorem 5.1

Table 5.2 The effects of the reductions on complexity

	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8
2.1a	≥	=	=	>				
2.1b	>							
2.2	=	=	≥	≥	≥	≥	≥	>
3.1	=	=	≥	=	=	>		
3.2	=	=	≥	=	>			
3.3	≥	=	>					
3.4	=	=	=	=	>			
3.5	=	=	≥	=	=	>		
3.6	=	>						
3.9	=	=	=	=	>			
4.1	≥	≥	≥	≥	≥	≥	≥	>
4.2	≥	≥	≥	≥	≥	>		
4.3	≥	=	=	=	=	>		
4.4	≥	=	=	=	=	>		
4.5	≥	=	≥	=	>			
4.6	=	=	=	=	=	>		
4.7	=	=	=	=	=	=	>	
	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8

Specifications of Functions used to Prove Theorem 5.2

$f_3[S] = \sum_{n=d \in S} f_{3.1}[S,d]$

$f_{3.1}[S,(a_1:d_1,a_2:d_2,\cdots,a_k:d_k)] = \sum_{i=i}^{k} f_{3.1}[S,d_i] + \prod_{i=i}^{k}(1+f_{3.2}[S,d_i])$

$f_{3.1}[S,(t_1:d_1|t_2:d_2|\cdots|t_k:d_k)] = \sum_{i=i}^{k} f_{3.1}[S,d_i]$

$f_{3.1}[S,\{d\}] = f_{3.1}[S,d]$

$f_{3.1}[S,<d>] = f_{3.1}[S,d]$

$f_{3.1}[S,n]$ = if n is a basic type or recursive in S then 0 else $f_{3.2}[S,expr[S,n]]$

where $expr[\{n_1=d_1,n_2=d_2,\cdots,n_k=d_k\},n_i] = d_i$

$f_{3.2}[S,(a_1:d_1,a_2:d_2,\cdots,a_k:d_k)] = \prod_{i=i}^{k}(1+f_{3.2}[S,d_i])$

$f_{3.2}[S,(t_1:d_1|t_2:d_2|\cdots|t_k:d_k)] = k + \sum_{i=i}^{k} f_{3.2}[S,d_i]$

$f_{3.2}[S,\{d\}] = 0$

$f_{3.2}[S,<d>] = 0$

$f_{3.2}[S,n]$ = if n is a basic type or recursive in then 0 else $f_{3.2}[S,expr[S,n]]$

$f_4[S] = \sum_{n=d \in S} f_{4.1}[S,d]$

$f_{4.1}[S,(a_1:d_1,a_2:d_2,\cdots,a_k:d_k)] = \prod_{i=i}^{k} f_{4.1}[S,d_i]$

$f_{4.1}[S,(t_1:d_1|t_2:d_2|\cdots|t_k:d_k)] = \prod_{i=i}^{k} f_{4.1}[S,d_i]$

$f_{4.1}[S,\{d\}] = f_{4.1}[S,d]$

$f_{4.1}[S,<d>] = f_{4.1}[S,d]$

$f_{4.1}[S,n] = 1 + f_{4.2}[S,n]$

$f_{4.2}[S,n]$ = if n is a basic type or recursive in S then 0 else $f_{4.1}[S,n]$

5.3.4. Proof of the Finite Church-Rosser Property

The given replacement system is finite Church-Rosser. This is shown by using Theorem 2.2 from Sethi's work on replacement systems [Set74] and the previous theorem of this section. Sethi's theorem states

> A replacement system $R=(S,\Rightarrow,=)$ is finite Church-Rosser if and only if R is finite and has properties P1 and P3.
>
> P1: If $u=v$ and $u\Rightarrow w$
> then there exist y and z such that $y=z$, $w\Rightarrow^{*}y$, and $v\Rightarrow^{*}z$.
>
> P3: If $u\Rightarrow w$ and $u\Rightarrow z$
> then there exist y and z such that $y=z$, $w\Rightarrow^{*}y$, and $z\Rightarrow^{*}z$.

Theorem 5.3

The replacement system of this section is finite Church-Rosser.

Proof

Finiteness is shown in Theorem 5.1. In order to conclude that the system is finite Church-Rosser, it is sufficient to show that Sethi's P1 and P3 properties hold.

Sethi's P1 property holds since if two schemas are structurally equivalent and one reduces then the other has an almost identical reduction. The result of applying these two similar reductions is two schema that are structurally equivalent. This follows from the fact that in every reduction, the names of tuple attributes and union domain is unimportant. The name are not preconditions of the reductions and have no significant influence on the results of the reduction.

Showing the P3 property requires a case by case analysis of each reduction with itself and each other reduction. Table 5.3 on the next page summarizes the analysis. An entry gives the codes for each pair of reductions. Each code represents a style of proof. A numeric code is used when a proof style applies to many cases. A proof

two codes, one proof is almost complete and the other handles some special cases. Following the table is a section that explains the meaning of each code.

Table 5.3 Cases in the proof of Sethi's P3 property

	1a	1b	2	1	2	3	4	5	6	9	1	2	3	4	5	6	7
2.1a	6																
2.1b	6	5															
2.2	7	1	5														
3.1	6	6	7	3													
3.2	6	6	7	3	3												
3.3	6	6	7	c	d	6											
3.4	6	6	7	3	e	6	1										
3.5	6	6	7	5	3	6	1	5									
3.6	1	1	1	2	2	f	1	2	1								
3.9	1	1	g	1	6	1	1	1	1	1							
4.1	2	1	1	2	2	2	2	2	1	1	1						
4.2	2	4	7	2,5	2	2,6	2	2,5	h	1	2	2,5					
4.3	1	4	7	1	1	6	1	1	1	3	2	2	1				
4.4	1	4	7	1	1	6	1	1	1	3	2	2	1	1			
4.5	1	4	7	3	2,3	a	b	3	1	3	2	2	3	1	5		
4.6	6	6	7	3	1	6	1	1	2	1	2	2	3	1	1	5	
4.7	6	6	7	1	1	6	1	1	1	1	2	2	1	1	3	3	1
	1a	1b	2	1	2	3	4	5	6	9	1	2	3	4	5	6	7

Cases applying to many pairs of reductions

1 The reductions always commute since they apply to different expressions or types.

2 The reductions either commute or the one with the bigger scope absorbs the other. R_1 absorbs R_2 if applying R_2 then R_1 is the same as applying R_1 only. For instance, the reduction *eliminate a valueless type* absorbs any reduction that is applicable to a sub-expression of the type's definition.

3 The reductions either commute or almost commute. R_1 and R_2 almost commute if applying R_1 then R'_2 is isomorphic to applying R_2 then R'_1 where R' is same type of reduction as R applied to a slightly different expression.

For instance in the expression (*a*:(*c*:*real*,*d*:*real*),*b*:*singleton*), the reductions *remove a singleton attribute from a tuple* and *include a subordinate tuple's attributes* almost commute. The expression reduces to (*a*:(*c*:*real*,*d*:*real*)) by removing the singleton attribute from the superordinate tuple and then to (*sub_c*:*real*,*sub_d*:*real*) by including the subordinate tuple's attributes. On the other hand, the expression also reduces to (*sub_c*:*real*,*sub_d*:*real*,*sup_b*:*singleton*) by including the subordinate tuple's attributes and then to (*sub_c*:*real*,*sub_d*:*real*) by removing the singleton attribute from the tuple.

4 One reduction is the separation of a type and the other deals with simplifying an expression referring to a valueless type. The reductions usually commute. If they do not, the same result is achieved by doing either reduction and then the following series of reductions.

1. For each type, simplify any expression referring to a valueless types by doing all the 4.2, 4.3, 4.4, and 4.5 reductions possible.

   After these reductions, each valueless type is of the form $n_1 = n_2$ and any other type does not refer to a valueless type.

2 Repeat eliminating any unreferenced valueless type as long as any exist.

After these reductions, any remaining valueless types are separable. This follows because if some type is not separable then there is some other type that has two type that directly reference it. Since each valueless type has one reference out and at least one in, each must have exactly one in.

3. Separate each valueless type.

After these reductions, all valueless types are not referenced by any other type.

4. Eliminate each valueless type.

5 The reductions either commute or produce schemas that are structurally equivalent. For instance, the reductions *simplify a single-attribute tuple* and *include a subordinate tuple's attributes* have isomorphic effects on the expression (*a*:(*d*:*real*),*b*:*real*,*c*:*integer*). After simplifying the single-attribute tuple, it reduces to (*a*:*real*,*b*:*real*,*c*:*integer*). After including the subordinate tuple's attributes, the original expression reduces to the structurally equivalent expression (*sub_d*:*real*,*sup_b*:*real*,*sup_c*:*integer*).

6 One of the reductions makes duplicate copies of sup-expressions. For instance, the reduction *change a tuple with a union to a union of tuples* makes duplicate copies of the sub-expression for the other attributes of the tuple. Separation and substitution also make duplicate copies of expressions.

An expression-duplicating reduction and most other reductions are either commutative or distributive. When done first, the expression-duplicating reduction gives a schema that requires the other reduction be done to each of the duplicate expressions to get a structurally-equivalent result to doing the other reduction followed by the expression-duplicating reduction.

In the expression (*a*:(*d*:*real*),*b*:*real*,*c*:(*t*1:*integer*|*t*2:*string*)), changing the tuple to a union makes two copies of the expression (*d*:*real*). Therefore, the reduction *simplify*

a single-attribute tuple must to be done to both copies if it is done after the reduction *change a tuple with a union to a union of tuples* to get the same result as doing the elimination once before the change.

7 The reduction *replace two isomorphic types by one type* eliminates one of two duplicate type definitions. This reduction commutes with other reductions except when the other reduction changes one of the duplicate definitions. In that case, there is a distributive situation similar to general case 6 above. If the other reduction is done first, it must be done to both types before the replacement can be done.

For instance, in the schema $\{n1=(a:real),n2=(a:real)\}$ the result of replacing either type with the other and then simplifying the single-attribute tuple is structurally equivalent to simplifying the single-attribute tuple from both types and then eliminating either type.

Cases applying to only one pair of reductions

a One reduction drops a domain on a valueless type from a union and the other changes a tuple with a union to a union of tuples. The reductions commute or almost commute except when the union is subordinate to the tuple. In most of those cases, dropping the domain and then changing the tuple to a union gives a schema that is structurally equivalent to the one from changing the tuple to a union, simplifying a tuple with the empty domain, and then dropping a domain from a union. However, if the original union has only two domains, dropping a domain on a valueless type gives a schema that is structurally equivalent to one from changing the tuple to a union, simplifying a tuple with an empty domain, and dropping a domain on a valueless type from a union.

For example, when the type n is valueless in a database schema, the expression $(a1:real,a2:(t1:n|t2:string|t3:real))$ reduces to $(a1:real,a2:(t2:string|t3:real))$ or to

$(t1{:}(a1{:}real, a2{:}n)|t2{:}(a1{:}real, a2{:}string)|t3{:}(a1{:}real, a2{:}real))$. Both of these reduce to $(t2{:}(a1{:}real, a2{:}string)|t3{:}(a1{:}real, a2{:}real))$. The first reduces directly but the second expression must first be reduced to the expression $(t1{:}n|t2{:}(a1{:}real, a2{:}string)|t3{:}(a1{:}real, a2{:}real))$.

b One reduction drops a domain on a valueless type from a union and the other eliminates a union inside a set. These commute or almost commute except when both modify the same union. In that case, the elimination of the union followed by replacing the set on a valueless type by a singleton and then dropping the singleton attribute out of the tuple has the same effect as dropping the valueless domain from the union before eliminating the union.

For illustration, when the type n is valueless in a database schema, the expression $\{(t1{:}real|t2{:}real|t3{:}n)\}$ reduces to $(t1{:}\{real\}, t2{:}\{real\}, t3{:}\{n\})$. This expression reduces to $(t1{:}\{real\}, t2{:}\{real\}, t3{:}singleton)$, and then to $(t1{:}\{real\}, t2{:}\{real\})$. On the other hand, the same result is produced if the original expression is reduced to $\{(t1{:}real|t2{:}real)\}$ and then to $(t1{:}\{real\}, t2{:}\{real\})$.

When the original union has two domains, eliminating the valueless domain eliminates the union. A structurally-equivalent schema is achieved by eliminating the union, simplifying a set on a valueless domain, removing a singleton attribute from a tuple and then simplifying a single-attribute tuple.

c One reduction includes a subordinate tuple's attribute and the other changes a tuple with a union to a union of tuples. The general case 6 applies except when the subordinate tuple is the one to be changed by the union. In that case, including the subordinate tuple's attributes and then changing the new tuple to a union gives a structurally-equivalent schema to the one produced by changing the subordinate tuple to a union, changing the superordinate tuple to a union and including a subordinate tuple's attributes in each tuple created by the second change.

For instance, the expression (*a*:*string*,*a*2:(*a*21:*real*,*a*22:(*t*1:*string*|*t*2:*real*))) is reduced to (*a*:*string*,*a*21:*real*,*a*22:(*t*1:*string*|*t*2:*real*)) by including the subordinate tuple's attributes. This expression can then be reduced to (*t*1:(*a*:*string*,*a*21:*real*,*a*22:*string*)|*t*2:(*a*:*string*,*a*21:*real*,*a*22:*real*)).

The same result is achieved when the original expression is reduced to (*a*:*string*,*a*2:(*t*1:(*a*21:*real*,*a*22:*string*)|*t*2:(*a*21:*real*,*a*22:*real*))). This reduces to (*t*1:(*a*:*string*,*a*2:(*a*21:*real*,*a*22:*string*))|*t*2:(*a*:*string*,*a*2:(*a*21:*real*,*a*22:*real*))), then to (*t*1:(*a*:*string*,*a*21:*real*,*a*22:*string*)|*t*2:(*a*:*string*,*a*2:(*a*21:*real*,*a*22:*real*))), and finally to (*t*1:(*a*:*string*,*a*21:*real*,*a*22:*string*)|*t*2:(*a*:*string*,*a*21:*real*,*a*22:*real*)).

d One reduction includes a subordinate union's domains and the other changes a tuple with a union to a union of tuples. The general case 6 applies except when the superordinate union is the one that is to change the tuple. In that case, including the subordinate union's domains followed by changing the tuple to a union gives the same result as changing the tuple to a union of tuples, changing the new tuple with the subordinate union to a union and then including that new union's domains in the union created by the first reduction.

For instance, the expression (*a*1:*string*,*a*2:(*t*1:*real*|*t*2:(*t*21:*string*|*t*22:*real*))) is reduced to (*a*1:*string*,*a*2:(*t*1:*real*|*t*21:*string*|*t*22:*real*)) by including a subordinate union's domains. Changing the tuple to a union reduces this expression to (*t*1:(*a*1:*string*,*a*2:*real*)|*t*21:(*a*1:*string*,*a*2:*string*)|*t*22:(*a*1:*string*,*a*2:*real*)).

The same result is also achieved when the original expression is reduced to (*t*1:(*a*1:*string*,*a*2:*real*)|*t*2:(*a*1:*string*,*a*2:(*t*21:*string*|*t*22:*real*))). This reduces to (*t*1:(*a*1:*string*,*a*2:*real*)|*t*2:(*t*21:(*a*1:*string*,*a*2:*string*)|*t*22:(*a*1:*string*,*a*2:*real*))), and then to (*t*1:(*a*1:*string*,*a*2:*real*)|*t*21:(*a*1:*string*,*a*2:*string*)|*t*22:(*a*1:*string*,*a*2:*real*)).

e One reduction includes a subordinate union's domains and the other eliminates a union inside a set. These commute or almost commute except when the superordi-

nate union is the union to be eliminated. Then the result of including the subordinate union's domain and eliminating the union is the same as eliminating both unions in turn and then including a subordinate tuple's attributes.

For example, by including a subordinate union's domains, the expression $\{(t1{:}real|t2{:}(t21{:}string|t22{:}real))\}$ is reduced to $\{(t1{:}real|t21{:}string|t22{:}real)\}$. This expression can then be reduced to $(t1{:}\{real\},t21{:}\{string\},t22{:}\{real\})$. When the original expression is reduced to $(t1{:}\{real\},t2{:}\{(t21{:}string|t22{:}real)\})$ and then to $(t1{:}\{real\},t2{:}(t21{:}\{string\},t22{:}\{real\}))$, the final result is the same.

f One reduction eliminates simple recursion in a type and the other changes a tuple with a union to a union of tuples. These commute or almost commute except when changing the tuple to a union puts the type in an inappropriate form for the elimination of all simple recursion. In that case, a sequence of changing the tuple to a union, including a subordinate union's domains and eliminating recursion usually has the same result as eliminating recursion, changing the tuple to a union and then including a subordinate union's domains. Changing the tuple to a union is not needed if the original tuple had two attributes. Including a subordinate union's domains is not needed if the original type had only one recursive domain.

To illustrate this argument, the simply recursive type definition

$$n=(t1{:}(a1{:}n,a2{:}real,a3{:}(t2{:}real|t3{:}string))|t4{:}integer)$$

is considered. Changing the tuple to a union reduces this type to $n=(t1{:}(t2{:}(a1{:}n,a2{:}real,a3{:}real)|t3{:}(a1{:}n,a2{:}real,a3{:}string))|t4{:}integer)$. This type is no longer in an appropriate form but reduces to one that is. That form $n=(t2{:}(a1{:}n,a2{:}real,a3{:}real)|t3{:}(a1{:}n,a2{:}real,a3{:}string)|t4{:}integer)$ reduces to $n=(sequence{:}<(t2{:}(a2{:}real,a3{:}real)|t3{:}(a2{:}real,a3{:}string))>,choice{:}integer)$. On the other hand, the reduction that eliminates all simple recursion produces $n=(sequence{:}<(a2{:}real,a3{:}(t2{:}real|t3{:}string))>,choice{:}integer)$. Then changing

this tuple to union gives the same type produced by the first series of reductions.

g One reduction splits an unreferenced union and the other replaces two isomorphic types by one type. These usually commute except when the unreferenced union is one of the isomorphic types. In that case, replacing two isomorphic types followed by a series of reductions gives the same result as splitting the unreferenced union followed by a series of reductions. The series following the replacement is zero or more substitutions that make the new union unreferenced and then a reduction to split into a set of types. On the other hand, the series to follow the splitting reduction of one of two isomorphic types is zero or more substitutions that make the other union unreferenced, a splitting of that union and then a series of the reduction *replace two isomorphic types by one type*. One replacement is required for each domain in the two original unions.

h One reduction eliminates all simple recursion in a type and the other simplifies a tuple with the empty domain. These usually commute except when the tuple with the empty domain is a recursive domain of the type. Simplifying the tuple eliminates all simple recursion in some cases. When it does not, the two reductions can be done in either order then dropping a domain from the appropriate union gives isomorphic results.

Simplify the valueless tuple eliminates all simple recursion in cases where the the valueless tuple is the only recursive domain. After eliminating all simple recursion, a structurally equivalent schema can be produced by simplifying the tuple, simplifying a sequence of a valueless type, removing a singleton attribute from a tuple, and simplifying a single-attribute tuple.

5.3.5. Improving the Replacement System

Probably, some improvements could be made to the replacement system but these would require intense analysis. The proof of the Church-Rosser property requires eighteen new sub-cases to add a single new reduction. Also, a new reduction might require more restrictions on the application of existing reductions. For example, the restriction on separation is required so that both it and elimination of simple recursion could be included.

Using inverse transformations has the potential of improving the replacement system. In particular, substitution has been restricted so the inverse reduction, *extraction*, would be useful. Given the database schema

$$\{n1=(a1{:}real, a2{:}(t1{:}string \mid t2{:}\{n2\})), n2=(t1{:}string \mid t2{:}\{n2\})\},$$

extracting $n2$'s definition from $n1$ would give

$$\{n1=(a1{:}real, a2{:}n2), n2=(t1{:}string \mid t2{:}\{n2\})\}.$$

Since $n1$ refers to $n2$ in both schemas, the second schema is clearly superior to the first because $n1$'s definition is simpler.

There are several reasons why extraction is not included in the replacement system. To produce a finite replacement system, a restriction on extraction would have to be found that guaranteed that an infinite sequence of extractions and substitutions did not exist. To produce a Church-Rosser replacement system, two problems would have to be overcome. The first schema above illustrates one of these. Its definition of $n1$ reduces to $n1=(t1{:}(a1{:}real, a2{:}string) \mid t2{:}(a1{:}real, a2{:}\{n2\}))$. If that reduction is done then an extraction is not possible. For the improved replacement system to be Church-Rosser, the definition of extraction must be general enough to detect any *hidden* extractions. It is not obvious that this detection is computable.

Assuming that it is possible to detect hidden extractions there is another serious problem. To demonstrated this problem, the schema

$\{n1=(a1{:}string, a2{:}real, a3{:}\{n2\}, a4{:}\{n3\}),$
$n2=(a1{:}string, a3{:}\{n2\}), n3=(a1{:}string, a4{:}\{n3\})\}$

Either the definition of $n2$ or $n3$ can be extracted from $n1$ but never both. If the replacement system is Church-Rosser, some arbitration of this conflict by the system is required. Since neither is intrinsically superior to the other, it would be extremely difficult to incorporate extraction.

5.4. Interesting Results

This section contains some results obtained by studying the consequences of having a finite Church-Rosser replacement system for normalizing database schemas.

5.4.1. Properties of Normal Form Schemas

A normal form schema has the following properties.

1. Each type has values.
2. If a type refers to another type, the other type is recursive.
3. If a type's highest-level constructor is a union then the type is recursive. This follows immediately from the point above. Recall that reduction 3.9 splits an unreferenced union type.
4. A union constructor is the highest-level constructor of a type or the constructor of the elements of a sequence. In other words, a union constructor never forms a sub-expression of any tuple, set, or union constructor.
5. A tuple constructor never forms a sub-expression of a tuple constructor.

5.4.2. The Expressive Power of the Union Constructor

For any database schema, the replacement system produces a corresponding normal form schema that is informationally equivalent in the allowed manner to the original schema. This section presents a theorem and a startling conjecture about the expressive power of the union type constructor. The theorem states that any database schema is informationally equivalent in the allowed manner to one with each type defined by an expression that has no union type constructors or is the union of expressions that have no union type constructors. The conjecture is that every schema is informationally equivalent in the allowed manner to one with each type defined by an expression that has no union type constructors what so ever. If the conjecture is correct, all of the explicit unions in any database schema can be replaced by the implicit union of all types in some equivalent schema.

Theorem 5.4

For any schema S, there exist a schema S' such that $S \equiv_A S'$ and all types of S' are either of the form $n = d$ where d has no union constructors or $n = (t_1 : d_1 | t_2 : d_2 | \cdots | t_k : d_k)$ where none of the d_i's has a union constructor.

Proof (by construction)

Given any schema S, construct an equivalent schema by the following algorithm.

1 Use the replacement system to find a normal form schema $\overline{S}$ with $\overline{S} \equiv_A S$.

2 Construct S' from $\overline{S}$ by replacing each sub-expression of the form $<(t_1:d_1|t_2:d_2|\cdots|t_k:d_k)>$ by the equivalent expression

$$\begin{aligned}(t_1:&<(t_1:d_1,t_2:<(t_2:d_2,t_3:<(t_3:d_3,\cdots,t_{k-1}:<(t_{k-1}:d_{k-1},t_k:<d_k>)>\cdots)>)>)>,\\ t_2:&<(t_2:d_2,t_3:<(t_3:d_3,\cdots,t_{k-1}:<(t_{k-1}:d_{k-1},t_k:<d_k>)>\cdots)>)>,\\ t_3:&<(t_3:d_3,\cdots,t_{k-1}:<(t_{k-1}:d_{k-1},t_k:<d_k>)>\cdots)>,\\ &\cdots\\ t_{k-1}:&<(t_{k-1}:d_{k-1},t_k:<d_k>)>,\\ t_k:&<d_k>).\end{aligned}$$

To see that $S' =_A \overline{S}$, starting with the replacement expression, apply many ($O(k^2)$) transformations to get the original expression with the union. The transformations *combine two sequences in a tuple* (Section 4.3.10) and *include a subordinate union's domains* are the only ones required.

Since $S' =_A \overline{S}$ and $\overline{S} =_A S$, the transitivity property gives $S' =_A S$.

Conjecture

For any schema S, there exist a schema S' such that $S =_A S'$ and all types of S' are of the form $n = d$ where d has no union constructors.

Argument

For any schema examined, it is possible to construct an equivalent schema that has no union constructors.

Given any schema S, an outline of the method to produce S' follows.

1 Use the replacement system to find a normal form schema $\overline{S}$ that is equivalent to S.

2 Repeat as long as possible.

a Do an appropriate finite sequence of substitutions that make some type simply recursive.

b Normalize the schema produced by the substitutions.

3 While the form of the database schema is

$$\{n_1 = d_1, n_2 = d_2, \cdots, n_i = (t_1 : d_{i_1} | t_2 : d_{i_2} | \cdots | t_k : d_{i_k}), \cdots, n_j = d_j\},$$

modify it to

$$\{n_1 = \hat{d}_1, n_2 = \hat{d}_2, \cdots, n_{i_1} = \hat{d}_{i_1}, n_{i_2} = \hat{d}_{i_2}, \cdots, n_{i_k} = \hat{d}_{i_k}, \cdots, n_j = \hat{d}_j\}$$

where $\hat{d}$ is d with each sub-expression n_i replaced by $(t_1 : n_{i_1} | t_2 : n_{i_2} | \cdots | t_k : n_{i_k})$.

4 Normalize the schema from 3.

5 In the manner described in Theorem 5.4, eliminate from each type any expressions of the form $<(t_1 : d_1 | t_2 : d_2 | \cdots | t_k : d_k)>$.

Each of the steps maintains informational equivalence in the allowed manner. The output of the final step does not have any union constructors since every different type of them is eliminated. The vagueness of step 2 is the reason that this conjecture is not a theorem. The word *appropriate* appears in step 2a because schemas exists where inappropriate substitutions can be repeated indefinitely.

5.4.3. The Equivalence Problems

The various equivalence relations naturally induce problems to be studied. Determining if two arbitrary database schemas are structurally equivalent is one of these and is called the *structural equivalence* problem. By relating this problem to two variations of the problem of determining if two directed graph are isomorphic, it can be shown that the structural equivalence problem is NP-complete. ([AHU74] discusses the NP-completeness of various problems including some graph isomorphism ones.)

Theorem 5.5

The structural equivalence problem is NP-hard.

Proof

The unlabeled directed graph isomorphism problem is reduced to the structural equivalence problem. This shows the later problem is NP-hard since the former problem is known to be NP-hard. The reduction is done by constructing a schema from a graph such that two graphs are isomorphic if and only if their respective schemas are structurally equivalent.

From any directed graph, construct a schema as follows.

For each node i in the graph, assign a unique type name n_i.

If the node i has arcs to the nodes $i_1, i_2, \cdots$, and i_k,

then define a type $n_i = (a: <n_i>, n_{i_1}: \{n_{i_1}\}, n_{i_2}: \{n_{i_2}\}, \cdots, n_{i_k}: \{n_{i_k}\})$ in the schema.

Clearly, two graphs are isomorphic if and only if their respective schemas are structurally equivalent.

Normal-form equivalence

Since the replacement system of the last section has the finite Church-Rosser properties, another definition of equivalence is possible. Two schemas S_1 and S_2 are *normal-form* equivalent if their normal forms are structurally equivalent.

The proof of Theorem 5.5 transforms a graph to a schema in normal-form. Therefore, the problem of determining normal-form equivalence is also NP-hard. In fact, the proof transforms a graph to a schema in normal form with respect to a hypothetical replacement system that uses all of the transformations of Sections 4.2 4.3 and 4.4. Thus, the problem of determining normal-form equivalence would probably be NP-hard regardless of the definition of the replacement system.

Theorem 5.6

The structural equivalence problem is NP-complete.

Proof

The structural equivalence problem is NP-hard from the previous theorem and it is now reduced to an NP-complete problem to show that it is NP-complete. The structural equivalence problem is reduced to the labeled directed graph isomorphism problem by constructing a graph from a schema such that two schemas are structurally equivalent if and only if their respective graphs are isomorphic.

From a schema, construct a graph as follows.

The graph has a node for each basic type of the data model. The node is labeled with the name of the basic type.

For each defined type in the schema, the graph has:

a node labeled with the literal "name"

a set of nodes and edges determined by the expression that defines the type.

an edge from the node for the type to the primary node determined by the expression that defines the type.

The primary node for the expression $(a_1{:}d_1, a_2{:}d_2, \cdots, a_k{:}d_k)$ is labeled "tuple" and has edges to the primary nodes determined by each of d_i's. The other types of expressions determine nodes and edges in a similar manner.

Clearly, two schemas are structurally equivalent if and only if their respective graphs are isomorphic.

Chapter 6

Conclusions

The first section of this chapter reviews the important results contained in the thesis. It is followed by a section containing some proposals for further work.

6.1. Review of Results

It was hoped that this thesis would be a complete treatment of informational issues of the semilattice data model. Although the thesis falls short of that ambitious goal, it contains many significant results.

The first of these results is a method of constructing a context-free grammar from a semilattice database schema. The language of the grammar has a strong relationship to the domain of the semilattice schema. The language consists of one or more of the notations for each value in the domain. The corollary to this result is that for any type in any schema, it can be decided if the type has the empty domain. Knowledge of valueless types is required by the corrective information-preserving transformations that are described in Section 3.4.

The use of a context-free grammar to describe the domain of a schema is appropriate since the domains of some schemas can not be described by regular grammars. Theorem 3.2 shows that the notation for the schema $\{n{=}(t1{:}(a{:}n,b{:}real)|t2{:}string)\}$ is not a regular set. For any schema of the appropriate nature, a proof of similar style can show that its notation is not a regular set. Most schemas with recursive types have the appropriate nature although some with valueless types do not.

Recall that information-preserving transformations are required by the semilattice data model. Their description is the first step in giving the model the ability to make applications independent of the conceptual schema. The transformations give a method of showing that two schemas are informationally equivalent. The

transformations also supply the required methods of transforming data values. Chapter 4 describes almost twenty basic information-preserving transformations. The descriptions specify when and how a schema is transformed. They also describe how to produce two functions. One function is from the values of the original schema to the values of the other. The other function is from the values of the other schema to the values of the original one. When these functions are composed one way, they are the identity function on the domain of one of the schema. When composed the other way, they are the identity on the other domain. It is felt that all of these data transformation functions can be replaced by queries once a query language exists. This will allow the use of a well-known, stronger definition of equivalence.

Some of the transformations have been documented elsewhere. The transformations *remove simple recursion from a type, combine similar domains of a union* and *combine two sequences in a tuple* are first reported in this thesis. The corrective transformations are also original work. Collectively, they eliminate all parts of a schema that contribute no information.

The major result of this thesis is the second replacement system of Chapter 5. The system incorporates most of the information-preserving transformations of Chapter 4. A series of examples is given to demonstrate the reasons for excluding some transformations and for restricting others. It is shown that the replacement system has the finite Church-Rosser property. In order for the system to have this property, involved definitions are needed for the reduction that separates a type and the reduction that removes all simple recursion from a type. Although it was hoped that no reduction with an involved definition would be used, these two were accepted because of their value. These reductions greatly increase the normalizing power of the replacement system.

The existence of a finite Church-Rosser replacement system allows other results.

It gives a simple method to produce from a given semilattice database schema, a schema that has few or no union constructors and is informationally equivalent in the allowed manner. The technique is given in the proof of Theorem 5.4. That theorem is significant in its own right and hopefully will simplify the task of formulating and proving other theorems about semilattice database schemas.

Showing that two schemas are structurally equivalent is an NP-complete problem. This is shown in the proofs of Theorems 5.5 and 5.6. Although this means that no simple general method of determining equivalence of schemas is possible, a heuristic technique work reasonably well in most cases. In linear time, a profile of each type of a schema is developed. Then only the isomorphisms that map types to those with an identical profile are considered. Typically, the set of isomorphisms is small or empty.

Fortunately, an unexpected and significant result is in the thesis. A new, simple technique of showing finiteness of a replacement system is described in Section 5.3.2. This technique has good potential for general application because of its modular style This facilitates quick development, understanding and modification of a finiteness proof in the same manner that modularity aids software development.

6.2. Proposals for Further Work

Many possibilities for further work exist. When a query language of the semilattice data model exists, Chapter 4 should be rewritten. The schema equivalence definition of [AAB82] should be used and the data transformation functions should be rewritten as queries. Also, work could be done with constrained schemas since many data models allow constraints on schemas. Additional type constructors could also be incorporated. Other data models permit types that are multi-sets, unique sequences, or partitions.

The replacement systems in this thesis have involved transformations and relatively simple elements. Despite its complexity, the second replacement system does

not make some obvious and desirable reductions. Perhaps by considering systems with more involved elements, a simplier system that has greater normalizing power can be developed.

Other methods of normalization should be considered. This is especially important if the goal is to produce a *nice* equivalent schema rather than a simple equivalent schema. From the outset, the method of normalization sought was one that required a finite Church-Rosser replacement system. In retrospect, it is clear that such a method of normalization has disadvantages as well as the advantages mentioned in the introduction. For instance, in order to have the Church-Rosser property, the second replacement system of Chapter 5 uses some transformations rather than their inverses. Although these transformations give the appropriate behavior to the replacement system, the result is that the normal form of some schemas is not concise and intuitive.

Proving the conjecture of Section 5.4.2 is the most interesting proposal for further work. That conjecture is that any semilattice database schema is informationally equivalent in the allowed manner to one that has no union type constructors. The argument for the conjecture describes a method of transforming a schema to an equivalent one with the desired property. Although this method worked in every case considered, its description is vague and requires further work. How is an "appropriate sequence of substitutions" determined? Closely related to this problem is the need to show that "as long as possible" is not forever in any case.

References

[AbB84] Serge Abiteboul and Nicole Bidoit, Non First Normal Form Relations to Represent Hierarchically Organized Data, *Proceedings of the Third ACM Symposium on Principles of Database Systems*, 1984, pp. 191-200.

[AbH84] Serge Abiteboul and Richard Hull, IFO: A Formal Semantic Database Model, *Proceedings of the Third ACM Symposium on Principles of Database Systems*, 1984, pp. 119-132.

[AHU74] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading, MA, 1974.

[AhU79] Alfred. V. Aho and Jeffrey D. Ullman, Universality of Data Retrieval Languages, *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, January 1979, pp. 110-120.

[ACO85] Antonio Albano, Luca Cardelli and Renzo Orsini, "Galileo: A Strongly-Typed, Interactive Conceptual Language", *ACM Transactions on Database Systems*, Vol. 10, No. 2, June 1985, pp. 230-260.

[AMM83] Hiroshi Ariswa, K. Moriya and T. Miura, Operations and the Properties of Non-First-Normal-Form Relation Databases, *Proceedings of the Nineth Conference on Very Large Data Bases*, 1983, pp. 197-203.

[Arm84] William W. Armstrong, *A Semilattice Database System*, Department of Computing Science, University of Alberta, Edmonton, 1984. (unpublished).

[Arm86] William W. Armstrong, *Outline of the Semilattice Database System*, Department of Computing Science, University of Alberta, Edmonton, 1986. (in preparation).

[AtP82] Paolo Atzeni and D. Stott Parker, Jr., Assumptions in Relational Database Theory, *Proceedings of the ACM Symposium on Principles of Database Systems*, 1982, pp. 1-9.

[AAB82] Paolo Atzeni, Giorgia Ausiello and Carlo Batini, "Inclusion and Equivalence Between Relational Database Schemata", *Theoretical Computer Science*, Vol. 19, No. 2, 1982, pp. 267-285, North-Holland Publishing Co..

[BMS81] Catriel Berri, A. O. Mendelzon, Y. Sagiv and Jeffrey D. Ullman, "Equivalence of Relational Database Schemes", *SIAM Journal of Computing*, Vol. 10, No. 2, 1981, pp. 352-370.

[Bob85] Kenneth Bobey, *Conceptual Level Specification of the Semilattice Model of Data*, Department of Computing Science, University of Alberta, Edmonton, 1985. (unpublished).

[Bor80] Sheldon A. Borkin, *Data Models: A Semantic Approach for Database Systems*, MIT Press, Cambridge, Massachusetts, 1980.

[Cod70] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks", *Communications of the ACM*, Vol. 13, No. 6, June 1970, pp. 377-387.

[Cod72] E. F. Codd, Further Normalization of the Database Relational Model, in *Data Base Systems*, Randall Rustin (ed.), Prentice Hall, Englewood Cliffs, NJ, 1972, pp. 33-64.

[Con85] Tim Connors, Equivalence of Views by Query Capacity, *Proceedings of the Fourth ACM Symposium on Principles of Database Systems*, 1985, pp. 143-148.

[DGK82] Umeshwar Dayal, Nathan Goodman and Randy H. Katz, An Extended Relation Algebra with Control Over Duplicate Elimination, *Proceedings of the ACM Symposium on Principles of Database Systems*, 1982, pp. 117-123.

[DDQ78] Peter J. Denning, Jack B. Dennis and Joseph E. Qualitz, *Machines, Languages, and Computation*, Prentice Hall, Englewood Cliffs, NJ, 1978.

[Der82] Nachum Dershowitz, "Ordering For Term-Rewriting Systems", *Theoretical Computer Science*, Vol. 17, 1982, pp. 279-301.

[FST85] Patrick C. Fischer, Lawrence V. Saxton, Stan J. Thomas and Dirk Van Gucht, "Interaction Between Dependencies and Nested Relational Structures", *Journal of Computer and System Sciences*, Vol. 31, No. 2, 1985, pp. 343-354.

[FoV82] J. Foisseau and F. R. Valette, A Computer Aided Design Data Model: FLOREAL, in *File Structures and Databases for CAD*, Jose Encarnacao and F.-L. Krause (ed.), North-Holland Publishing Co., New York, NY, 1982, pp. 315-330. (Proceedings of the IFIP WG 5.2 Working Conference on ... 1981).

[GaV85] Shashi K. Gadia and Jay H. Vaishnav, A Query Language for a Homogeneous Temporal Database, *Proceedings of the Fourth ACM Symposium on Principles of Database Systems*, 1985, pp. 51-56.

[GuF86] Dirk Van Gucht and Patrick C. Fischer, Some Classes of Multilevel Relational Structures, *Proceedings of the Fifth ACM Symposium on Principles of Database Systems*, 1986, pp. 60-69.

[GHW85] J. V. Guttag, J. J. Horning and J. M. Wing, *Larch in Five Easy Pieces*, Digital Equipment Corp., Palo Alto, California, 1985.

[Har84] Martin Hardwick, Extending the Relational Database Model for Design Applications, *Proceedings of the Twenty-First Design Automation Conference*, 1984, pp. 110-116.

[HNC84] Lee Hollaar, Brent Nelson, Tony Carter and Raymond A. Lorie, The Structure and Operation of a Relation Database System in a Cell-Oriented Integrated Circuit Design System, *Proceedings of the Twenty-First Design Automation Conference*, 1984, pp. 117-125.

[HoU79] John E. Hopcroft and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, Reading, MA, 1979.

[HuL78] G. Huet and D. S. Lankford, On the Uniform Halting Problem for Term Rewriting Systems, Report 283, INRIA, Le Chesnay, France, 1978.

[HuY82] Richard Hull and Chee K. Yap, The Format Model: A Theory of Database Organization, *Proceedings of the ACM Symposium on Principles of Database Systems*, 1982, pp. 205-211.

[HuY84] Richard Hull and Chee K. Yap, "The Format Model: A Theory of Database Organization", *Journal of the ACM*, Vol. 31, No. 3, July 1984, pp. 518-537.

[Hul84] Richard Hull, Relative Information Capacity of Simple Relational Database Schemata, *Proceedings of the Third ACM Symposium on Principles of Database Systems*, 1984, pp. 97-109.

[JaS82] G. Jaeschke and Hans-Jorg Schek, Remarks on the Algebra of Non First Normal Form Relations, *Proceedings of the ACM Symposium on Principles of Database Systems*, 1982, pp. 124-138.

[KTT83] Y. Kamabayashi, K. Tanaka and K. Takeda, "Synthesis of Unnormalized Relations Incorporating More Meaning", *Information Science,* Vol. 29, 1983, pp. 201-247.

[KiK82] Hiroyuki Kitagawa and Tosiyasu L. Kunii, APAD: An Application-Adaptable Database System, in *Data Base Design Techniques II: Physical Design and Implementations*, Vol. 133, S. B. Yao and Tosiyasu L. Kunii (ed.), Springer Verlag, 1982, pp. 320-344. Presented at IBM Symposium on Database Engineering, Tokyo, 1979.

[KuV84] Gabriel M. Kuper and Moshe Y. Vardi, A New Approach to Database Logic, *Proceedings of the Third ACM Symposium on Principles of Database Systems*, 1984, pp. 86-96.

[KuV85] Gabriel M. Kuper and Moshe Y. Vardi, "On the Expressive Power of the Logical Data Model", *Proceedings of the SIGMOD Annual Meeting, SIGMOD Record,* Vol. 15, No. 2, 1985, pp. 180-187.

[Lie82] Y. Edmund Lien, "On The Equivalence of Database Models", *Journal of the ACM,* Vol. 29, No. 2, 1982, pp. 333-362.

[Lor82] Raymond A. Lorie, Issues in Databases for Design Applications, in *File Structures and Databases for CAD*, Jose Encarnacao and F.-L. Kruase (ed.), North-Holland Publishing Co., New York, NY, 1982, pp. 213-222. (Proceedings of the IFIP WG 5.2 Working Conference on ... 1981).

[Mai83] David Maier, *The Theory of Relation Databases*, Computer Science Press, Rockville, Maryland, 1983.

[MaW85] Zohar Manna and Richard Waldinger, *The Logical Basis for Computer Programming*, Addison Wesley, Reading, MA, 1985.

[MUS82] Hideo Matsuka, Sakae Uno and Masaaki Sibuya, Specific Requirements in Engineering Data Base, in *Data Base Design Techniques II: Physical Design and Implementations*, Vol. 133, S. B. Yao and Tosiyasu L. Kunii (ed.), Springer Verlag, 1982, pp. 345-356. Presented at IBM Symposium on Database Engineering, Tokyo, 1979.

[OzY85] Zehra Meral Ozsoyoglu and Li-Yan Yuan, A Normal Form for Nested Relations, *Proceedings of the Fourth ACM Symposium on Principles of Database Systems*, 1985, pp. 251-260.

[RND77] Edward M. Reingold, Jurg Nievergelt and Narsingh Deo, Combinatorial Algorithms Theory and Practice, *Prentice Hall*, Englewood Cliffs, NJ, 1977, pp. 361.

[ScP82] Hans-Jorg Schek and P. Pistor, Data Structures for an Integrated Database, *Proceedings of the Eighth Conference on Very Large Data Bases*, 1982, pp. 197-207.

[Set74] Ravi Sethi, "Testing for Church-Rosser Property", *Journal of the ACM,* Vol. 21, No. 4, October 74, pp. 671-679.

[ShW85] Arie Shoshani and Harry K. T. Wong, "Statistical and Scientific Database Issues", *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 10, October 1985, pp. 1040-1047.

[Sin86] Ajit Singh, Access Methods for a Semilattice Database Management System, M. Sc. Thesis, Department of Computing Science, University of Alberta, Edmonton, 1986.

[SAH84] Michael Stonebraker, Erika Anderson, Eric Hanson and Brad Rubstein, "Quel as a Data Type", *Proceedings of the SIGMOD Annual Meeting, SIGMOD Record,* Vol. 14, No. 2, 1984, pp. 208-214.

[TsL82] Dionysios C. Tsichritzis and Fredrick H. Lochovsky, *Data Models*, Prentice Hall, Englewood Cliffs, NJ, 1982.

[Ull82] Jeffrey D. Ullman, The U.R. Strikes Back, *Proceedings of the ACM Symposium on Principles of Database Systems*, 1982, pp. 10-22.

[Zan83] Carlo Zaniolo, "The Database Language GEM", *Proceedings of the SIGMOD Annual Meeting, SIGMOD Record,* Vol. 13, No. 3, 1983, pp. 208-214.

[ZaH85] Robert Zara and David R. Henkle, Building a Layered Database for Design Automation, *Proceedings of the Twenty-Second Design Automation Conference*, 1985, pp. 645-651.